VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER SYSTEMS

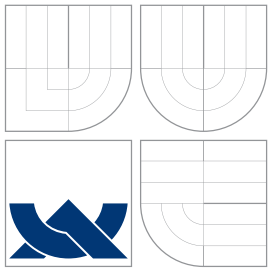
PACKET CLASSIFICATION ALGORITHMS

DISERTAČNÍ PRÁCE
PHD THESIS

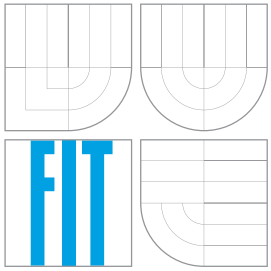
AUTOR PRÁCE
AUTHOR

Ing. VIKTOR PUŠ

BRNO 2012



VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER SYSTEMS

ALGORITMY KLASIFIKACE PAKETŮ

PACKET CLASSIFICATION ALGORITHMS

DISERTAČNÍ PRÁCE
PHD THESIS

AUTOR PRÁCE
AUTHOR

Ing. VIKTOR PUŠ

VEDOUCÍ PRÁCE
SUPERVISOR

Prof. Ing. VÁCLAV DVOŘÁK, DrSc.

BRNO 2012

Abstrakt

Tato práce se zabývá klasifikací paketů v počítačových sítích. Klasifikace paketů je klíčovou úlohou mnoha síťových zařízení, především paketových filtrů – firewallů. Práce se tedy týká oblasti počítačové bezpečnosti. Práce je zaměřena na vysokorychlostní sítě s přenosovou rychlostí 100 Gb/s a více. V těchto případech nelze použít pro klasifikaci obecné procesory, které svým výkonem zdaleka nevyhovují požadavkům na rychlost. Proto se využívají specializované technické prostředky, především obvody ASIC a FPGA. Neméně důležitý je také samotný algoritmus klasifikace. Existuje mnoho algoritmů klasifikace paketů předpokládajících hardwarovou implementaci, přesto však tyto přístupy nejsou připraveny pro velmi rychlé sítě. Dizertační práce se proto zabývá návrhem nových algoritmů klasifikace paketů se zaměřením na vysokorychlostní implementaci ve specializovaném hardware. Je navržen algoritmus, který dělí problém klasifikace na jednodušší podproblémy. Prvním krokem je operace vyhledání nejdelšího shodného prefixu, používaná také při směrování paketů v IP sítích. Tato práce předpokládá využití některého existujícího přístupu, neboť již byly prezentovány algoritmy s dostatečnou rychlostí. Následujícím krokem je mapování nalezených prefixů na číslo pravidla. V této části práce přináší vylepšení využitím na míru vytvořené hashovací funkce. Díky použití hashovací funkce lze mapování provést v konstantním čase a využít při tom pouze jednu paměť s úzkým datovým rozhraním. Rychlost tohoto algoritmu lze určit analyticky a nezávisí na počtu pravidel ani na charakteru síťového provozu. S využitím dostupných součástek lze dosáhnout propustnosti 266 milionů paketů za sekundu. Následující tři algoritmy uvedené v této práci snižují paměťové nároky prvního algoritmu, aniž by ovlivňovaly rychlost. Druhý algoritmus snižuje velikost paměti o 11 % až 96 % v závislosti na sadě pravidel. Nevýhodu nízké stability odstraňuje třetí algoritmus, který v porovnání s prvním zmenšuje paměťové nároky o 31 % až 84 %. Čtvrtý algoritmus kombinuje třetí algoritmus se starším přístupem a díky využití několika technik zmenšuje paměťové nároky o 73 % až 99 %

Abstract

This thesis deals with packet classification in computer networks. Classification is the key task in many networking devices, most notably packet filters – firewalls. This thesis therefore concerns the area of computer security. The thesis is focused on high-speed networks with the bandwidth of 100 Gb/s and beyond. General-purpose processors can not be used in such cases, because their performance is not sufficient. Therefore, specialized hardware is used, mainly ASICs and FPGAs. Many packet classification algorithms designed for hardware implementation were presented, yet these approaches are not ready for very high-speed networks. This thesis addresses the design of new high-speed packet classification algorithms, targeted for the implementation in dedicated hardware. The algorithm that decomposes the problem into several easier sub-problems is proposed. The first subproblem is the longest prefix match (LPM) operation, which is used also in IP packet routing. As the LPM algorithms with sufficient speed have already been published, they can be used in out context. The following subproblem is mapping the prefixes to the rule numbers. This is where the thesis brings innovation by using a specifically constructed hash function. This hash function allows the mapping to be done in constant time and requires only one memory with narrow data bus. The algorithm throughput can be determined analytically and is independent on the number of rules or the network traffic characteristics. With the use of available parts the throughput of 266 million packets per second can be achieved.

Additional three algorithms (PFCA, PCCA, MSPCCA) that follow in this thesis are designed to lower the memory requirements of the first one without compromising the speed. The second algorithm lowers the memory size by 11 % to 96 %, depending on the rule set. The disadvantage of low stability is removed by the third algorithm, which reduces the memory requirements by 31 % to 84 %, compared to the first one. The fourth algorithm combines the third one with the older approach and thanks to the use of several techniques lowers the memory requirements by 73 % to 99 %.

Klíčová slova

Klasifikace paketů, počítačová bezpečnost, hardwarová akcelerace, FPGA

Keywords

Packet Classification, computer security, hardware acceleration, FPGA

Citation

Viktor Puš: Packet Classification Algorithms, disertační práce, Brno, FIT VUT v Brně, 2012

Packet Classification Algorithms

Declaration

I declare that this thesis is my original work and that I have written it under the supervision of prof. Ing. Václav Dvořák, DrSc. All sources and literature that I have used during elaboration of the thesis are correctly cited with complete reference to the corresponding sources.

.....
Viktor Puš
May 1, 2012

Acknowledgements

I would like to thank prof. Ing. Václav Dvořák, DrSc. for his supervision of this work and for his valuable suggestions. I would also like to thank other colleagues from the Departement of Computer Systems, Faculty of Information Technology, Brno University of Technology and from the Departement of Tools for Monitoring and Configuration, CESNET association. Special thanks go to Ing. Jan Kořenek, PhD., who has been my mentor for years and whose help with my research was far beyond his duties.
I would also like to thank my family and friends for their great support and patience.

This work has been partially supported by projects Optical National Research Network and Its New Applications MSM63483917201, TeamIT - Building Competitive Research Teams in IT CZ.1.07/2.3.00/09.0067, Secured, reliable and adaptive computer systems FIT-S-10-1, Advanced secured, reliable and adaptive IT FIT-S-11-1, CESNET Large Infrastructure LM2010005.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

With the rapid development of computer networks, security threats such as viruses and other attacks by hackers are also on the rise. Network security is studied and applied at various layers: direct filtering at the packet level, intrusion detection at the application level, traffic monitoring and detection of anomalous behavior of the whole network. Network traffic filtering has become one of the first steps in securing any network or computer. While the packet filter cannot detect and block all dangerous network traffic, it is still one of the most effective building blocks for any computer security system. It also often cooperates with the higher levels of network security. For example, anomaly detection system updates filtering rules based on the assessment of current threats.

The packet filtering system must perform packet reception, packet header parsing, packet classification and the action based on the instruction from the matching classification rule. Packet classification is the most complex part of the system, and it therefore determines the speed, and also mostly the cost of the system. This task is not only important in practical applications, but also interesting from the theoretical point of view, having implications in geometry and other fields [37].

As network speeds are increasing, the demand for high speed packet processing is also growing. While 10 Gb/s ports are commonly present in various networking devices, the 100 Gb/s technology is expected to be increasingly available in the near future. Standard for 100 Gb/s Ethernet was proposed as IEEE standard 802.3ba in 2008 and ratified in June 2010. This new standard is capable of transmitting one packet each 6.7 ns in each direction. Packet classification must be able to achieve this throughput, otherwise it would throttle the bandwidth.

Many algorithms for packet classification were proposed [24, 17, 23, 43, 31, 7, 44, 33, 47, 46], but the goal of 100 Gb/s throughput is either beyond the limits of the current technology, or requires excessive amount of high-speed (and thus expensive) memory.

The algorithms oriented on high speed use various methods of hardware acceleration. Application-specific Integrated Circuits (ASICs) and Field-Programmable Gate Arrays (FPGAs) are commonly used. Ternary Content-Associative Memories (TCAMs) can also be found in commercial devices in conjunction with ASICs of FPGAs. Due to high development cost and long time-to-market of ASICs and also the high cost and power consumption of TCAMs, FPGAs gain increasing popularity. Programmability of FPGAs make them suitable for the research in the field of packet classification.

Properties of each packet classification approach are defined by two main factors: The technology used, and the algorithm running on the selected device. While the technology is gradually improved by silicon vendors, and can be approximately predicted by the

application of Moore's law, new algorithms may bring significant improvements immediately. That's why the research of packet classification algorithms is important for practical applications.

Gupta [23] views packet classification as a geometrical problem of multi-dimensional search and therefore proposes a decision tree which reduces the state space until correct rule is found. This algorithm is further improved by Singh et al. [43] and Vamanan et al. [50].

Lakshman and Stiliadis [31] introduce problem decomposition into packet classification. Their Bit Vector algorithm performs set of independent searches on ranges, and then combines the results together. Bit Vector algorithm is improved by Baboescu and Varghese [7], Li et al. [33] and Song and Lockwood [44].

Srinivasan et al. [46] replace the direct geometrical representation of the packet classification problem by purely combinatorial construction of Cartesian product. They also use search on prefixes instead of ranges. Taylor and Turner [47] and Dharmapurikar et al. [17] further improve the idea of high-speed processing of the Cartesian product set.

Almost all algorithms deal with some trade-offs. In the case of packet classification, there is often a struggle between the speed and the size of required memory. Example of the algorithm which has the potential to achieve very high speed is the direct Cartesian product algorithm [46]. This algorithm, however, is practically unusable due to very large mapping table it uses. While improvements of this algorithm have significantly lower memory requirements, their speed is limited by more complex processing loop [47] or several accesses to the external memory [17].

This thesis proposes new packet classification algorithm tailored for high-speed applications, targeting 100 Gb/s networks. Its unique property is the throughput higher than 100 Gb/s, even in the worst case. This speed cannot be achieved by the current algorithms. The lack of guaranteed throughput is generally an issue of all current algorithms which is completely eliminated by the algorithm presented in this thesis.

The algorithm is intended for implementation in a hardware accelerator (ASIC or FPGA) and therefore it is designed with consideration of capabilities of these devices. All steps of the algorithm are either very simple arithmetic operations and memory accesses, or were shown to be able to run at the speeds required for 100 Gb/s network in ASIC or FPGA. The idea of problem decomposition introduced in recent literature [17] is also employed in this thesis. The algorithm is implemented as a processing pipeline, which brings higher speed by exploiting the parallelism inherent to ASIC and FPGA devices. The algorithm is built around the idea of finding fast direct mapping from the results of independent (and thus potentially parallel) prefix search engines to the correct rule number. Perfect (collision-free) hash function construction algorithm [13] is used to create such direct mapping.

In addition to exploiting inherent parallelism of hardware implementation, the algorithm achieves high speed by employing rather complex software precomputation phase for finding the mapping function which then leads to very simple evaluation of the mapping function in the hardware. Due to the fact that there is no loop in the process of classification, the throughput of the algorithm is perfectly deterministic and constant. Speed of the algorithm is comparable or better than the fastest known algorithms.

While the speed of the algorithm is excellent, it requires significant amount of memory for its data structures. This thesis therefore discusses the possibility of lowering memory requirements of the algorithm. Three following algorithms make use of empirically obtained facts about common properties and the structure of rule sets. They lower the memory

consumption of the original algorithm by application of several new optimization techniques. These methods add very simple logic to the processing pipeline of the algorithm in order to avoid the states which generate excessive requirements on the amount of memory in the first algorithm.

The next chapter introduces issues in computer network security and presents current approaches of dealing with them. After that, the problem of packet classification is described in more detail and necessary definitions of terms are provided. Chapter 4 presents several existing algorithms published by other authors. Chapter 5 analyzes these approaches and also presents basic properties of data sets used in this thesis for experiments. Four following chapters present four new packet classification algorithms. The first one aims to achieve high speed, the others improve its memory consumption. Chapter 10 compares the four presented algorithms to each other and also to some selected algorithms from Chapter 4. The last chapter concludes the thesis and sums up its contributions.

Chapter 2

Issues and Techniques of Computer Security

This chapter briefly introduces current issues in computer network security. Basic devices and techniques used to resolve those issues are then presented. Some other uses of packet classification (not necessarily directly related to security) are mentioned at the end of the chapter.

2.1 General Issues

Security of computer systems and networks has been an issue for a long time. With the rising number of computers connected to the Internet, this global network has become the primary way of spreading security threats. Current attackers are no longer harmless amateurs trying to prove their computer skills. More and more often network administrators must deal with professional attacks. Motivation of such attacks may include industry espionage, attempts to harm competition, or even military actions (the so-called cyber war).

As the attackers are using more and more sophisticated methods, their attempts can be divided into several groups:

- *Port scanning* is not a direct attack. This technique is used to find out which services are available at the device. This information may be used to guess the type of device (ie. the operating system running on the computer) and also to find a weak point in the system. Common port scanning techniques can be detected by finding a difference from a normal network traffic behavior pattern.

- *Denial of Service (DoS)*: This attack aims to send overwhelming number of requests to the system (open enormous number of TCP flows, for example), so the system is unable to respond to all of them. This affects also legitimate requests, so the availability of the system is lowered. Distributed version of this attack (DDoS) uses many computers to generate requests, so it is harder to reveal and block the source of the attack. (D)DoS can be detected by measuring the consumed resources (ie. CPU load) of the system under attack and also by finding anomalies in the network traffic.

- *Malware* is a software which masks its malicious behavior behind some useful function. After the trusting user installs the malware, the attacker is able to gain control over the computer. This control can be then used to steal private user data, send spam,

take part in DDoS or other criminal activity. Malware is detected and prevented on the hosts by antivirus software. Known malware can be detected by finding specific patterns in the packet data. Malware spreading can also be visible in the network behavior statistics.

- *Penetration Attack* is a direct attack, which uses known vulnerabilities of the system or password guessing to gain access to the system. After gaining the access, attacker can perform similar actions as in the malware-infected computer. Penetration attacks are often detected by searching for attack patterns in packets.

- *Man in the Middle*: This attack is possible if an attacker is able to intercept and alter the communication between two hosts. This ability can be used to compromise security of various services, such as email. Man in the middle attacks may be prevented by using strong encryption.

- *Combinations*: Most attacks use combination of several techniques. Recent example is the Chuck Norris botnet [52, 51]: Port scanning was used to find insecure ADSL routers, then the penetration attack (password guessing) was performed to gain access to these routers (creating the botnet), and finally, the botnet could be used to perform DDoS or man in the middle attack.

2.2 Firewall

Firewall is a networking device which examines the network traffic and treats the network data according to the pre-programmed filtering rules. The evolution of firewalls can be split into three generations:

- The simplest *stateless* firewall observes only the packet header. The filtering decision is based only on the filtering rules and on the information contained in a single packet header. The packet either passes the device or is discarded. The firewall state is not modified by packets (possible exception is logging).

- The *statefull* firewall also observes packet header, but in addition it contains the state memory for flows. The rule can check the flow state from the memory and the action associated to the rule can update the record in the state memory. Typical usage of this mechanism is the memory of open TCP flows: Packets from the outside network can not enter the secured network unless the TCP flow has been established from the trusted host in the secured network.

- The *application layer filter* or *proxy* works at the higher layers. It fully analyzes the communication and performs more complex actions. The example is web proxy which parses the HTTP protocol and filters URLs. Additional functionality of such proxy can be caching of popular web pages.

Packet classification addressed in this thesis is a stateless process and directly corresponds to the decision process of the stateless firewall. Each packet classification decision is based only on the filtering rules and the information from a single packet. This functionality is however the basis of the statefull firewall and is also often present in more advanced filtering devices. Therefore, the stateless filtering is still of great importance in firewalls.

The common use case of the firewall is at the borders of the network (Figure 2.1). In that case, the traffic flowing into or from the network is filtered by the rules, while the traffic inside the network is not filtered (is considered safe).

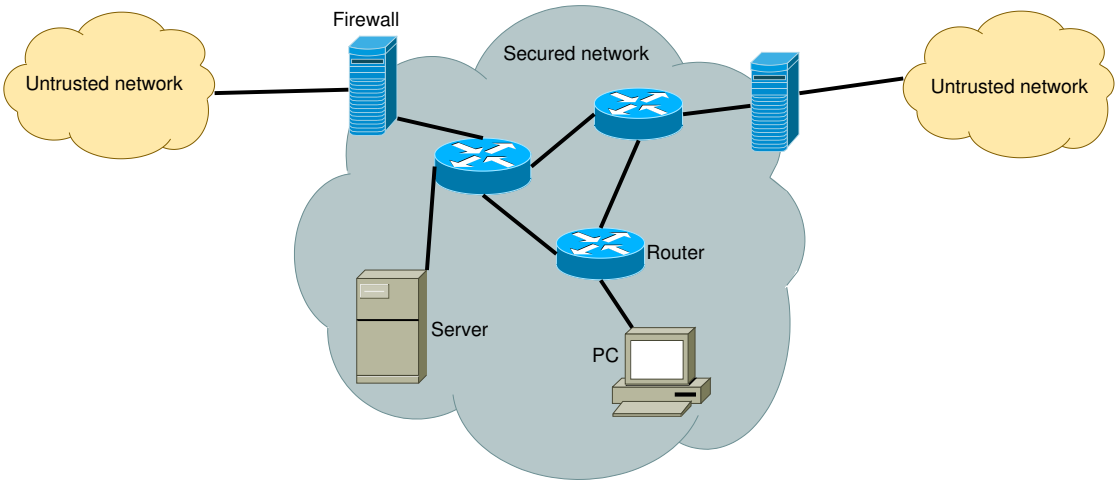


Figure 2.1: Use of firewalls at the network borders.

While the firewall can not provide complete protection of the network or host, it is the basic building block of the network security. Popular example of more complex network security structure is the demilitarized zone (DMZ). DMZ is a sub-network that contains and exposes an organization's external services to the untrusted network. In a network, the hosts most vulnerable to attack are those that provide services to users outside of the local area network, such as e-mail, web and Domain Name System (DNS) servers. Because of the increased potential of these hosts being compromised, they are placed into their own sub-network in order to protect the rest of the network if an intruder were to succeed in attacking some of them.

Hosts in the DMZ have limited connectivity to the internal network, although communication with other hosts in the DMZ and to the external network is allowed. This allows hosts in the DMZ to provide services to both the internal and external network, while an intervening firewall controls the traffic between the DMZ servers and the internal network clients. Two basic structures of a network with DMZ use one three-port firewall (Figure 2.2) or two two-port firewalls (Figure 2.3).

There are many software and hardware firewalls available, free and commercial. Software firewalls are sometimes integral part of operating systems, for example Microsoft Windows XP and newer [6]. Popular example of the free software firewall is the Berkley Packet Filter [35]. It provides the interface for sending and receiving packets, with the option of packet filtering. This mechanism is commonly used in Unix-like operating systems. Software firewalls are intended mainly for personal use, because their throughput is limited by the architecture and performance of personal computers.

On the other hand there are hardware firewalls, which can have the architecture optimized for packet filtering and therefore can achieve much higher throughput. Another advantage of having the firewall as a separate device is that the packet filtering does not add load to the processor of the protected computer. The example of commercial hardware firewall is the Cisco ASA 5500 Series [2]. The highest model from this series is claimed to support up to 2 000 000 simultaneous firewall connections and to achieve the throughput of

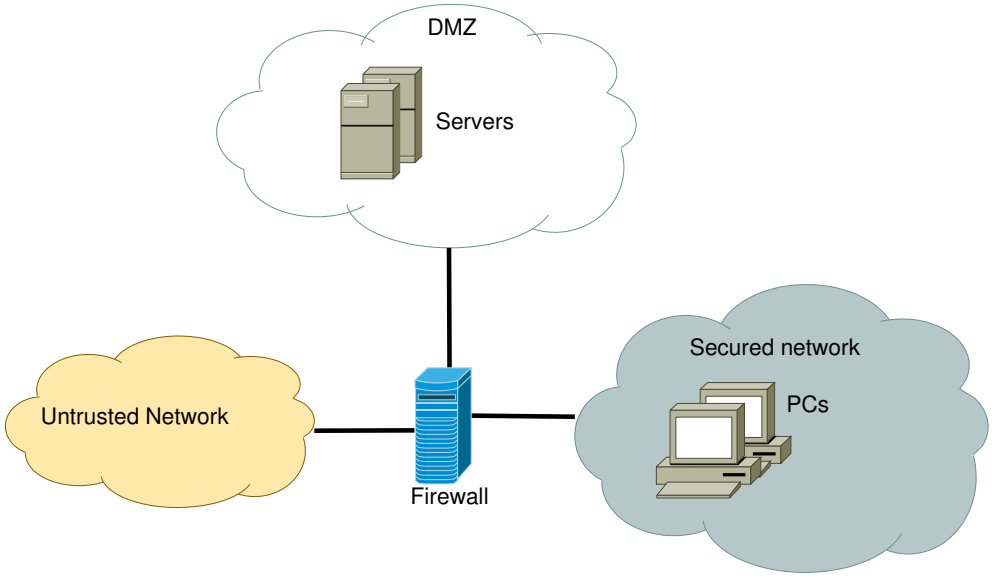


Figure 2.2: Network with demilitarized zone using one firewall.

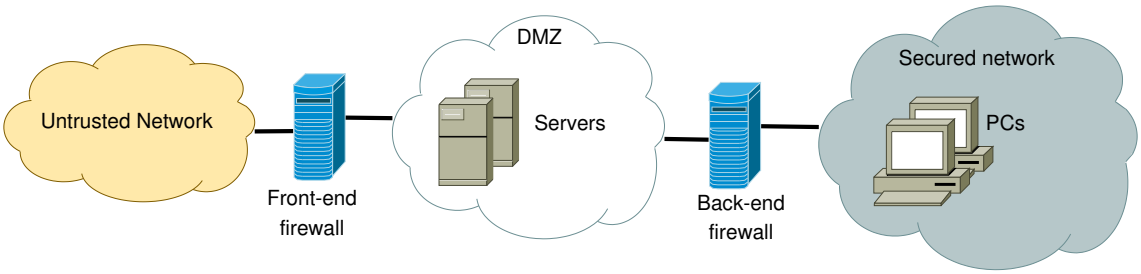


Figure 2.3: Network with demilitarized zone using two firewalls.

9 000 000 packets per second (for the shortest 64 B packets). The disadvantage of hardware firewalls is certainly the price.

From the view of the attack classification given in the previous section, firewall is not designed to prevent any specific type of attack. It is instead used to block sources of attacks detected by the methods that follow in the text. Firewalls are also used to enforce organization policy (for example: block the insecure FTP protocol by discarding all packets with port number 20 or 21).

2.3 Intrusion Detection System

The Intrusion Detection System (IDS) performs more detailed packet analysis than the firewall. It searches for the known patterns of attacks in the packet payload. The searching may be done by full analysis of the application protocol or by regular expression matching. In both cases, the aim is to understand (at least to a certain extent) the traffic at the application layer. This approach is particularly effective for detecting malware and penetration attacks.

The disadvantage of IDS is the need for quality database of threats, because IDS can only detect known attacks. The database often has the form of a set of regular expressions,

or expressions of even stronger language. Matching of these expressions to packet data has extreme demands on the computational power, and the research in this field is very actual [27, 30]. Popular implementations of IDS are Snort [5] and L7 Filter [1].

Intrusion prevention system (IPS) is an extension to IDS. It adds the *mitigation* functionality that follows after the detection. It is targeted not only to find and analyze security threats, but also to prevent them as soon as possible. IPS employs firewalls to block the malicious traffic. After the intrusion is detected, firewalls are quickly configured to block the source of the attack and thus the intrusion is mitigated.

2.4 Flow Monitoring and Anomaly Detection

Flow monitoring is a measurement technique for obtaining detailed insight into the network. It records information about flows – sets of packets with common properties (ie. addresses and ports) observed at some point in the network. There is a set of standards defining the measurement, transmission and storage of measured data [11, 49].

Querying tools provide various statistics about the network. Anomaly detection systems are often designed to work with these statistics. A model of normal network behavior is created and any bigger difference of the actual network state from the model is marked as anomaly. Not each anomaly is an attack, so the anomaly detection systems suffer with some rate of false positive alarms. The reported anomalies must often be checked by a human operator.

The flow monitoring and anomaly detection systems often work over the whole network (Figure 2.4). There are several measurement points, where probes passively observe the traffic and report the flow information to the flow collector. Measurement point may be placed also inside the network. Anomaly detection engine queries the collector, compares the network state to the model, and raises alarms.

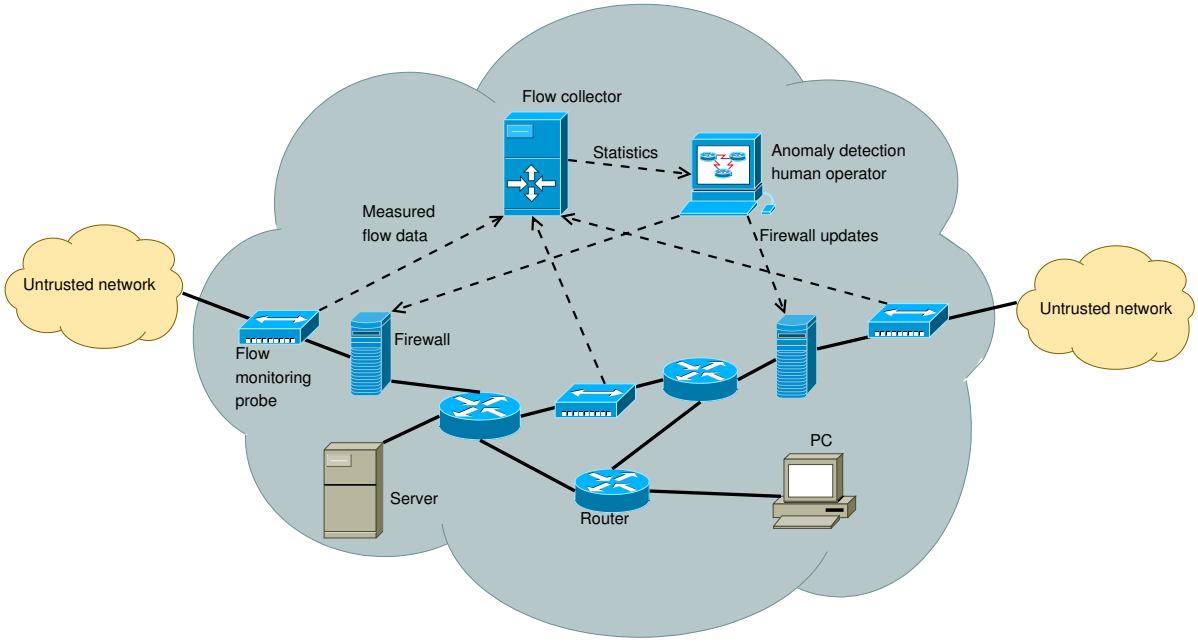


Figure 2.4: Anomaly detection system.

Flow monitoring and anomaly detection systems can detect unknown attacks, if such

attacks somehow change the network behavior. These systems were demonstrated to detect port scanning, (D)DoS attacks and malware spreading through the network [21].

Anomaly detection system itself only monitors and analyzes the network behavior. It is however often used in cooperation with firewalls. Based on the behavioral analysis, these firewalls are manually or automatically configured to block sources of suspicious traffic, as shown in Figure 2.4.

Other possible use of traffic filtering in anomaly detection is the measurement focusing. When a network anomaly is detected while performing the overall network measurements, the flow monitoring probes are configured to monitor only the traffic related to the anomaly, but with better detail. For example, the whole suspicious packets are directly transmitted from the probes to the collector for further detailed analysis.

2.5 Lawful Interception

Besides firewall, other common application of packet filtering is the Lawful Interception System (LIS). In this case, the Lawful Enforcement Agency (LEA) sends interception requests with the information about users whose communication must be intercepted and handed over to LEA. The LIS passively observes the traffic at the interception point (at the network border, for example). User IP address must be often dynamically derived from authorization protocols such as DHCP and RADIUS. The IP address of the target is then sent to the packet filter which sends the selected traffic to LEA for the purpose of analysis and evidence (Figure 2.5).

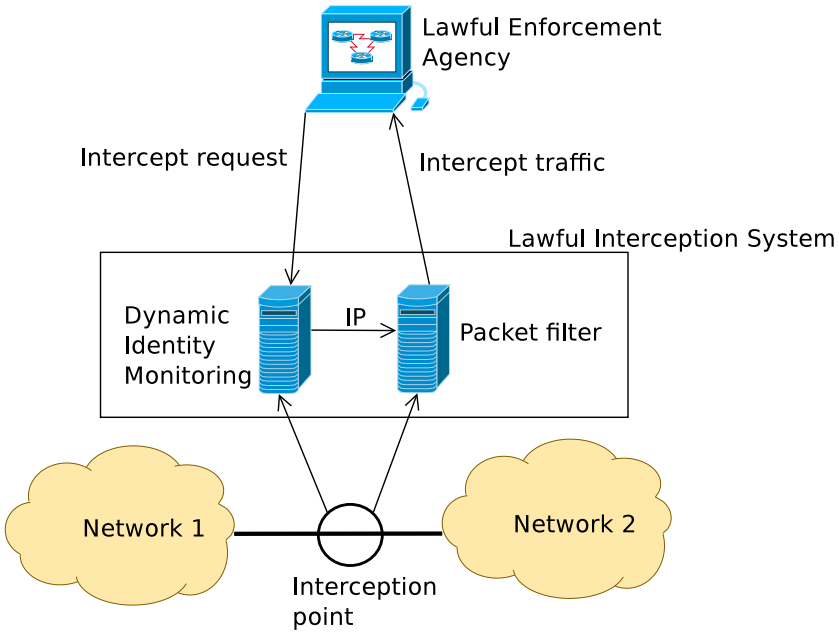


Figure 2.5: Lawful interception system.

Besides filtering, packet classification may be used for accounting and billing for the amount of transferred network data. In the task of ensuring the quality of services, packet classification is used to determine the priority of packets (Figure 2.6).

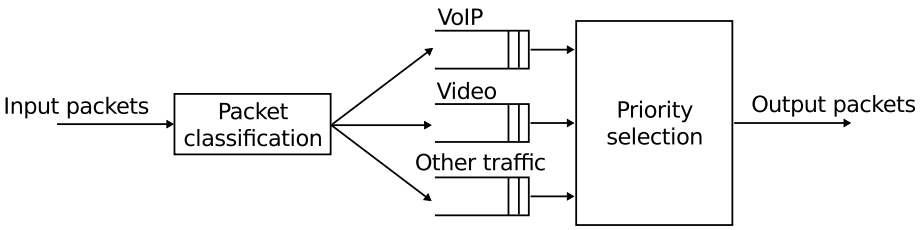


Figure 2.6: Packet classification used in ensuring quality of services.

Chapter 3

Formal Description of Packet Classification

This section defines terms and notions used in the following text. After the basic mathematical definitions are given, some additional information about other possible representations of the packet classification problem is presented. The end of the chapter describes practical aspects of a good packet classification algorithm.

3.1 Classification Dimension

Packet classification performs decision about each packet based on the values found in the packet header. Only some of packet header fields are important for classification, we call them *dimensions*. This term is used in geometry, and its meaning fits very well also for packet classification, because the problem of packet classification has direct geometrical representation (see Section 3.6).

Five dimensions are commonly used for packet classification:

- Internet Layer protocol's source and destination address. Internet Protocol version 4 (IPv4) or Internet Protocol version 6 (IPv6) are often used as Internet Layer protocols.
- Internet Layer protocol's next header. This dimension identifies the type of Transport Layer protocol.
- Transport Layer protocol's source and destination port number (if present). Transport Control Protocol (TCP) and User Datagram Protocol (UDP) are often used as Transport Layer protocols.

Definition 3.1 (Dimension size)**.** *Dimension size is the number of bits of the packet header field (16 for port number, 128 for IPv6 address, for example).*

3.2 Condition and Prefix

Each classification rule defines one *condition* for each dimension. There are several types of conditions:

- *Value*: Condition defines the exact value of packet header field.

- *Range*: Condition defines the range of allowed values of packet header field. Range is often used to specify port numbers.
- *Prefix*: Condition defines the exact value of several higher bits of packet header field. Only those bits are being matched. The remaining (lower) bits are not being matched, and therefore the packet header field may have any values at those bit positions. The number of matched bits is the *prefix length*. Prefixes are often used for IP addresses to match the whole subnet.

Definition 3.2 (Condition matching)**.** *Packet header field matches the condition c if it fits into the range of c.*

Definition 3.3 (Prefix covering)**.** *Prefix p_1* covers *prefix p_2 if range of p_2 is fully within the range of p_1. Every prefix covers itself.*

Definition 3.4 (Universal condition)**.** *Condition is universal, if it covers all possible values in its dimension. Universal condition is denoted* ANY *or *.*

3.3 Conversion of range to prefix

Range is the most universal type of condition, because each other type can also be expressed by range:

- *Value* is naturally a range with the identical lower and upper bounds.
- To convert a *prefix* into a range, lower and upper range bounds are obtained by setting non-matched bits to all zeros and ones respectively.

However, prefixes are often used for further processing because of their straightforward tree representation (see Section 3.4). To convert the range into prefixes, the longest possible prefixes are always used. For example, range $\langle 12; 15\rangle$ is always converted as binary prefix $11*$ and never as two prefixes $110*$ and $111*$.

Theorem 3.1. *Conversion of range to prefixes may result in up to $2s-2$ prefixes, where s is the dimension size. That is the case of the range $\langle 1; 2^s-2\rangle$.*

Proof. We are searching for integer range boundaries $a, b \in \langle 0; 2^s-1\rangle, b \geq a$, such that the range $\langle a; b\rangle$ generates the maximal number of prefixes. If both a and b are in the lower half $\langle 0; 2^{s-1}-1\rangle$ of the state space, then the maximal number of prefixes for the dimension size s is the same as for $s-1$, because lowering s by one and keeping a and b yields the same number of prefixes. It does not hold that increasing s also increases the number of prefixes. Similar situation occurs if both a and b are in the higher half $\langle 2^{s-1}; 2^s-1\rangle$ of the state space, because both halves of the state space are the same except for the first bit of their prefixes. We however later show that number of prefixes grows linearly with the dimension size. Therefore we assume that a is in the lower half of the state space $a \leq 2^{s-1}-1$ and b is in the higher half of the state space $b \geq 2^{s-1}$.

The problem is now reduced to finding $a \in \langle 0; 2^{s-1}-1\rangle$ such that the range $\langle a; 2^{s-1}-1\rangle$ is converted to the maximal number of prefixes. Finding $b \in \langle 2^{s-1}; 2^s-1\rangle$ such that the range $\langle 2^{s-1}; b\rangle$ is converted to the maximal number of prefixes is a symmetrical problem.

If a is in the higher half $\langle 2^{s-2}; 2^{s-1}-1\rangle$ of the new state space then the same number of prefixes can be obtained for the decremented s by moving a to the lower half of the

state space $a \leftarrow a - 2^{s-2}$ and removing the higher half, therefore lowering s yields the same number of prefixes. We however later show that the number of prefixes grows linearly, therefore we now continue with the assumption that a is in the lower half $\langle 0; 2^{s-2} - 1\rangle$ of the new state space.

The higher half $\langle 2^{s-2}; 2^{s-1} - 1\rangle$ of the new state space is a prefix and the problem is reduced again to finding $a \in \langle 0; 2^{s-2} - 1\rangle$ such that range $\langle a; 2^{s-2} - 1\rangle$ is converted to the maximal number of prefixes.

We get a recursive problem which continues until $s - x = 1$. Then a must be chosen from the range $\langle 0; 2^1 - 1\rangle = \langle 0; 1\rangle$. $a = 1$ is chosen, because $a = 0$ generates only one prefix for any range $\langle a; 2^y - 1\rangle$.

Each level of recursion generates one prefix, the search for a performs $s - 1$ recursive steps and therefore generates $s - 1$ prefixes. The search for b is symmetrical and generates also $s - 1$ prefixes. The whole state space can therefore generate up to $2s - 2$ prefixes. That happens for $a = 1, b = s^2 - 2$. □

Algorithm 3.1 is a general recursive procedure converting one range into several prefixes. The method recursively checks prefixes (starting with the prefix covering the whole dimension). If the currently examined prefix is fully within the range being converted, then this prefix is added to the result set and the recursion stops. If the currently examined prefix is outside the range being converted, the recursion stops. If the currently examined prefix partially covers the range being converted, it is split into halves and both halves are recursively examined by the same procedure.

Algorithm 3.1 Converting range to set of prefixes.

Input: Range bounds *low*, *high*, currently examined prefix bounds x_1, x_2.

1: This algorithm is first called with $x_1 = 0$ and $x_2 = 2^s - 1$.
2: **if** $\langle x_1; x_2\rangle \in \langle low; high\rangle$ **then**
3: Add prefix with bounds $\langle x_1; x_2\rangle$ to the result set.
4: **else**
5: **if** $x_1 \neq x_2 + 1$ and $((low \in \langle x_1; x_2\rangle)$ or $(high \in \langle x_1; x_2\rangle))$ **then**
6: $x_3 = x_1 + (x_2 - x_1)/2$
7: $x_4 = x_3 + 1$
8: Recursively call self with parameters *low*, *high*, x_1, x_3.
9: Recursively call self with parameters *low*, *high*, x_4, x_2.
10: **end if**
11: **end if**

Output: Set of all prefixes created at the line 3 of this algorithm.

3.3.1 Example

The left half of the Figure 3.1 shows an example conversion of one range to three prefixes. Grey vertical bars denote prefixes partially covered by the range. Recursion continues in these cases. Black bars denote prefixes fully covered by the range, while white bars mark prefixes not included in the range. Recursion stops when black or white bar is found. The right half of the Figure 3.1 shows the maximal number of prefixes generated by the range $\langle 1; 14\rangle$. The 4-bit dimension generates $2 \times 4 - 2 = 6$ prefixes.

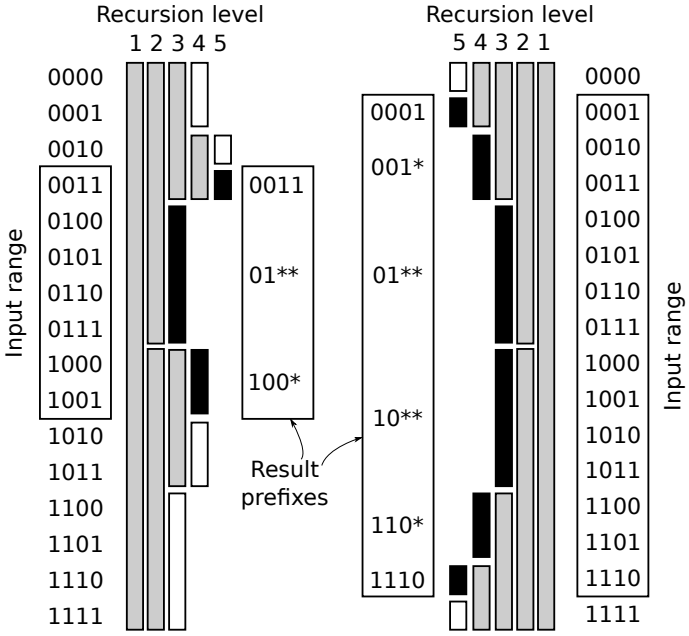


Figure 3.1: Example of converting range $\langle 3; 9\rangle$ to three prefixes and the example of maximal number of prefixes.

3.4 LPM and Trie

The Longest Prefix Match operation (LPM) is the search in the set of prefixes.

Definition 3.5 (LPM)**.** *From the set of prefixes with various lengths, the LPM operation finds the longest prefix which matches the input full-length value.*

Binary trie is a simple tree data structure for effective storage of prefixes. Every tree node contains pointers to two child nodes and to the result. Pointers may be null. Prefixes are stored in the structure of the tree, there is no need to store their values explicitly. Trie directly supports the LPM operation. The tree traversal is directed by the bits (from MSB to LSB) of the input value. If the bit being processed is 0, then the processing continues to the left child. If the bit is 1, then the right path is taken. The search ends if all input bits are processed of if null pointer is encountered. The last observed non-null pointer to the result determines the result of the LPM operation.

Figure 3.2 shows an example set of prefixes and the trie data structure for their storage and the longest prefix match operation. Valid prefixes (having non-null pointers to the result) are represented by black circles in the figure.

3.5 Rule and Rule Set

Definition 3.6 (Rule)**.** *Rule is an ordered* $(d+2)$*-tuple where* d *is the number of dimensions. It contains* d *conditions (one for each dimension), the rule priority and the rule action.*

Rule priority is expressed as integer number, where higher number means higher priority. Rules with the same priority are allowed in the rule set only if they do not overlap. Non-overlapping rules with the same priority are created when ranges are converted to prefixes. The *rule action* can contain instruction about how to treat the packet in the device.

Prefix	Name
10*	p1
1010*	p2
10101	p3
111*	p4

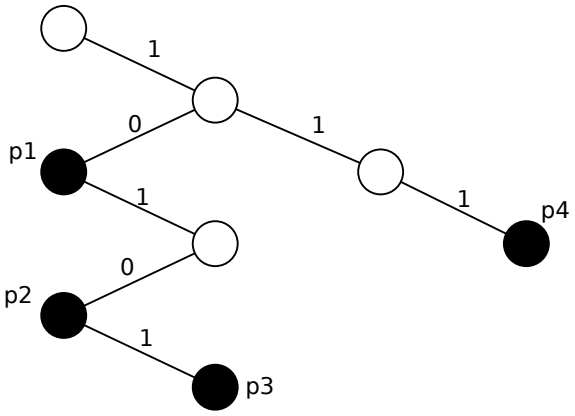


Figure 3.2: Example of binary trie with 4 stored prefixes.

Definition 3.7 (References to conditions)**.** *Rule r's condition for dimension d is denoted r.d.*

Definition 3.8 (Rule matching)**.** *Packet matches the rule if all packet header fields match the corresponding conditions of the rule.*

Rules are often expressed by some simple syntax, as shown in Figure 3.3.

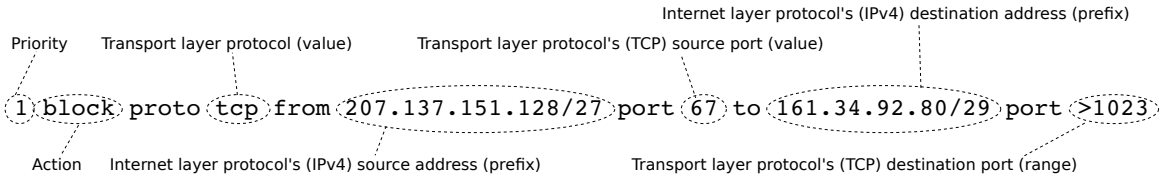


Figure 3.3: Example of rule written in human-readable syntax.

Definition 3.9 (Rule set)**.** *Rule set is a set of n rules.*

Definition 3.10 (Prefix set)**.** *Prefix set is the set of all prefixes found in one dimension of the rule set.*

Definition 3.11 (Universal rule)**.** *Universal rule is a rule which is matched by any packet (all its conditions are universal).*

The process of packet classification is the search in the set of rules. While one packet may match more rules in the rule set, the packet classification algorithm must return the rule with the highest priority that the packet matches. It can be assumed that the packet classification process *simulates* the linear search in the rule set ordered by priority, stopping at the first matching rule.

The process of obtaining packet header field values from the packet (header parsing) is out of scope of this thesis. Actions associated to rules are also not considered in this work. The rule number can be used to find appropriate action.

3.6 Geometrical Representation of Packet Classification

Packet classification may be viewed as a geometrical problem of searching in multi-dimensional discrete space. Each dimension (as defined in Section 3.1) may actually define one

dimension of the space of all possible packets. Each condition then defines an interval in this dimension and thus each rule defines one hyperrectangle (which is often formally defined as the Cartesian product of intervals in geometry). Each packet then defines one point of the space, possibly contained in several hyperrectangles (rules).

The process of packet classification from all rules containing (matching) the packet selects the one with the highest priority. There are packet classification algorithms which directly use the geometrical representation of packet classification [23, 43, 50].

Figure 3.4 and Table 3.1 show the example of the simple rule set with rules using only two dimensions of size three. In practice, the number of dimensions is typically five, and their sizes may be up to 128 bits for IPv6 addresses.

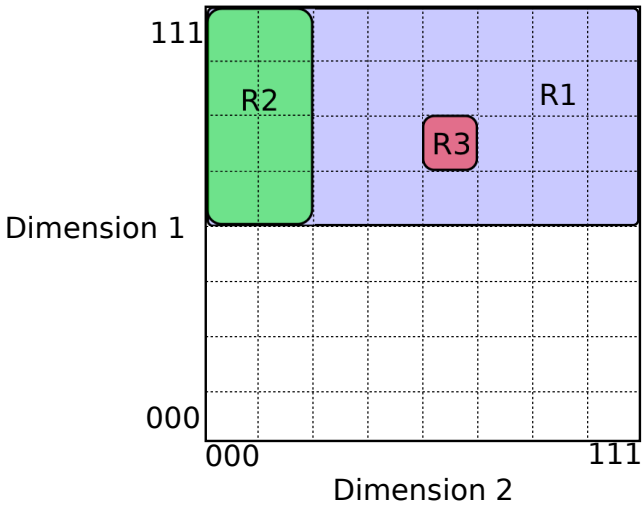


Figure 3.4: Geometrical representation of three rules in two three-bit dimensions.

Rule	Dimension 1	Dimension 2	Priority
R1	1*	*	1
R2	1*	00*	2
R3	101	100	3

Table 3.1: Example rule set. Asterisk (*) represents bits masked by prefix.

3.7 Properties of Packet Classification Algorithms

The classification algorithm must meet some requirements to be feasible in networking devices:

- **Speed.** The algorithm must run in real time, otherwise it would throttle the link bandwidth. The worst case is transmission of the shortest possible packets, because then the number of packets per time unit is highest. For example, 100 Gbps Ethernet transmits 148 809 523 packets of length 64 B per second in one direction. That is 6.72 ns/packet.

Formal measure is the time complexity, often measured as the worst case number of external memory accesses needed to classify one packet.

- **Memory.** The required amount of memory is often measured in bytes per rule. It has impact on the device cost, but may also affect speed, because smaller memories are often faster (i.e. static versus dynamic memory).
- **Area.** In case of ASIC or FPGA implementation, chip area is the important measure of the algorithm.
- **Other properties.** There are other measures for packet classification algorithms. For example support for incremental rule set updates, speed of such updates, latency, etc.

Chapter 4

Algorithms for Packet Classification

This chapter introduces current algorithms for packet classification. At the beginning, some very simple approaches are presented to illustrate that the problem of packet classification is not trivially solvable. Following sections describe some of the currently most important packet classification algorithms. Solutions based on geometrical representation, range matching and problem decomposition are presented, while the last group receives the greatest attention.

4.1 Straightforward Approaches

The simple algorithm is the linear search in rules. It requires only $O(n)$ space, but also $O(n)$ time (where n is the number of rules), which is impractical for most rule sets. However, some algorithms use the linear search in the reduced set of rules [23].

Another straightforward algorithm is the direct mapping into table. If all packet fields are concatenated into one wide word, this word can be used as an address to the precomputed table, which contains the correct rule number for each possible packet. It is clear that the size of such table is prohibitive for practical implementation, but this principle serves as a basis for some algorithms [22].

4.2 TCAM

Other rather simple approach is the use of Ternary Content-Associative Memory (TCAM). TCAM is a specialized memory organized in rows. Each row is composed of number of simple memory cells, which can be set to three different states: 0, 1 and *don't care*. The row matches the input word if memory cells with states 0 and 1 are equal to the corresponding bits of the input word. Cells in the don't care state are ignored during matching. TCAM is able to match all rows in parallel, which makes it very fast.

After converting range conditions to prefixes, all conditions of each rule can be expressed as a word containing ones, zeros and don't-cares. Each TCAM row stores one rule. To search in the rule set, one wide data word is created from packet header fields and sent to TCAM. The TCAM then performs parallel search in all rows to find the match.

It may seem that the TCAM performs the search in constant time $O(1)$. This is untrue, because the searching runs in n parallel branches (where n is the number of rules), and

therefore TCAM performs n searches to classify each packet. However, TCAMs are often used in commercial devices [43, 17]. While TCAMs certainly have their advantages, such as speed and method simplicity, they also have several disadvantages. The cost per bit of a high performance TCAM is about 15 times larger than a comparable SRAM (Static Random Access Memory) and they consume more than 50 times more power per access [17]. This gap between SRAM and TCAM cost and power consumption makes it worthwhile to continue to explore better algorithmic solutions.

4.3 Tree Based Algorithms

Hierarchical Intelligent Cuttings (Hi-Cuts) Algorithm [23] uses the geometrical representation of the packet classification problem. It constructs a decision tree where inner tree nodes divide the space by hyperplanes into several equally-sized subspaces. The division is performed until the number of rules in the subspace is lower then some defined threshold. After that, leaf node containing pointers to the remaining rules is created.

The subspace containing the packet is selected as a next node during the tree descent while searching for the rule. Leaf tree node containing pointers to several rules is searched sequentially. Figure 4.1 shows the example of the space cutting. Several improvements of the Hi-Cuts algorithm were published: HyperCuts [43] is able to cut the space in more than one dimension in each tree node. EffiCuts [50] allows to cut the space to non-equally sized parts.

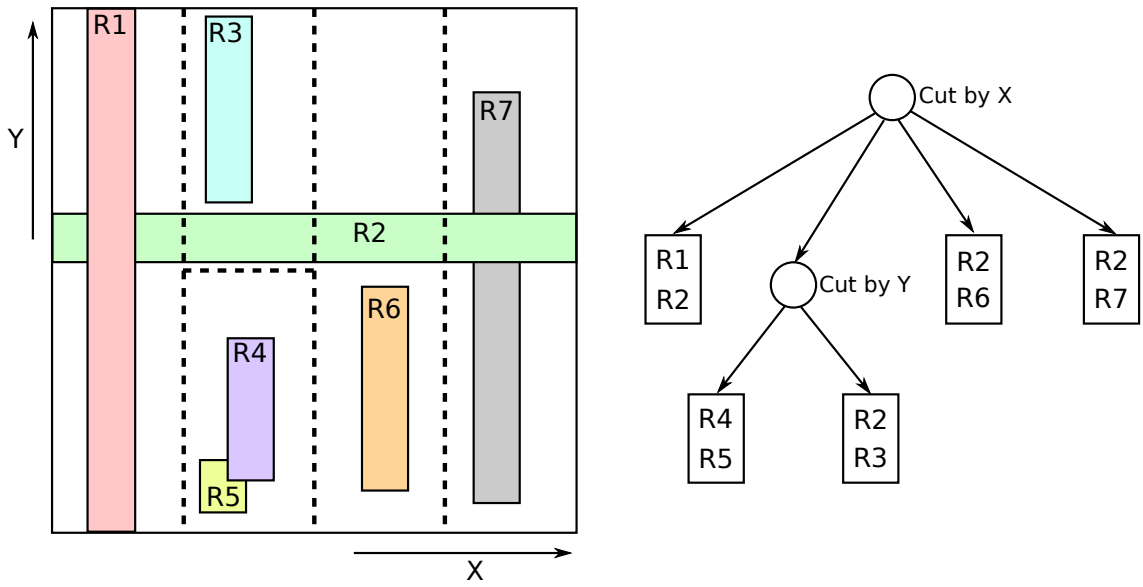


Figure 4.1: Example of two-dimensional space divided by Hi-Cuts hyperplanes and the corresponding Hi-Cuts tree.

The disadvantage of Hi-Cuts and also other algorithms derived from it is that the depth of a search tree depends on a particular rule set. Some rules be found earlier than at the bottom of the tree, but the worst case analysis must always take into account the longest tree branch. This results in different algorithm throughput for different rule sets.

4.4 Range Matching Algorithms

Another group of algorithms using geometrical representation of packet classification employs preprocessing of packet header fields by the range matching operation. The Bit Vector algorithm [31] projects all ranges of each dimension to the axis of that dimension. Each range creates up to two division points on the axis, therefore projecting all ranges results in no more than $2n$ division points and $2n+1$ non-overlapping ranges, where n is the number of ranges in that dimension. Each range is then assigned a bit vector containing one bit for each rule. Bits of the vector are assigned to one if the corresponding rule covers that range.

Figure 4.2 shows the example of three rules in two-dimensional space. The rules created 7 ranges on the horizontal axis and 5 ranges on the vertical axis. Each range then contains three-bit vector indicating presence of the range in each of the three rules.

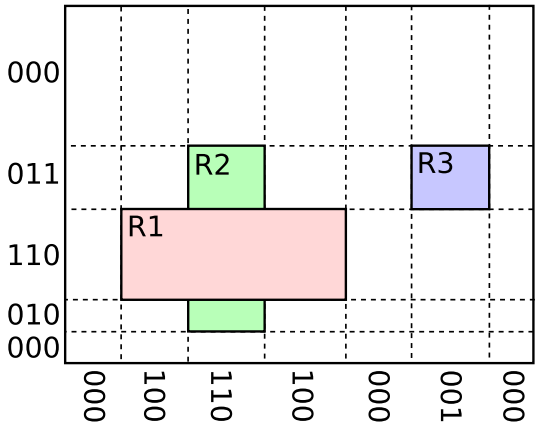


Figure 4.2: Example of ranges and associated bit vectors created by the Bit Vector algorithm.

The process of packet classification in the Bit Vector algorithm is shown in Figure 4.3. It first performs independent range search for all packet header fields. The range search is performed in the set of non-overlapping ranges obtained from the rule set and can be done by binary search in logarithmic time. The results of searches are bit vectors. Bitwise logical AND is then computed over all bit vectors to get the final bit vector, containing ones at the positions of matched rules.

The greatest disadvantage of the Bit Vector algorithm is the linear scaling of the bit vectors with the number of rules. Several algorithms were introduced to improve the original Bit Vector algorithm:

Aggregated Bit Vector algorithm [7] uses recursive aggregation of bit maps and rule rearrangement to improve the original algorithm significantly. The Aggregated and Folded Bit Vector algorithm [33] discards rule rearrangement and introduces a new concept of bit folding to further improve the Aggregated Bit Vector algorithm. The BV-TCAM architecture [44] combines the TCAM and the Bit Vector algorithm to effectively compress the data representations and boost throughput.

4.5 Recursive Classification

The Recursive Flow Classification (RFC) algorithm [22] is based on the method of direct mapping of packets into the table of rules. Instead of using all packet headers as an address

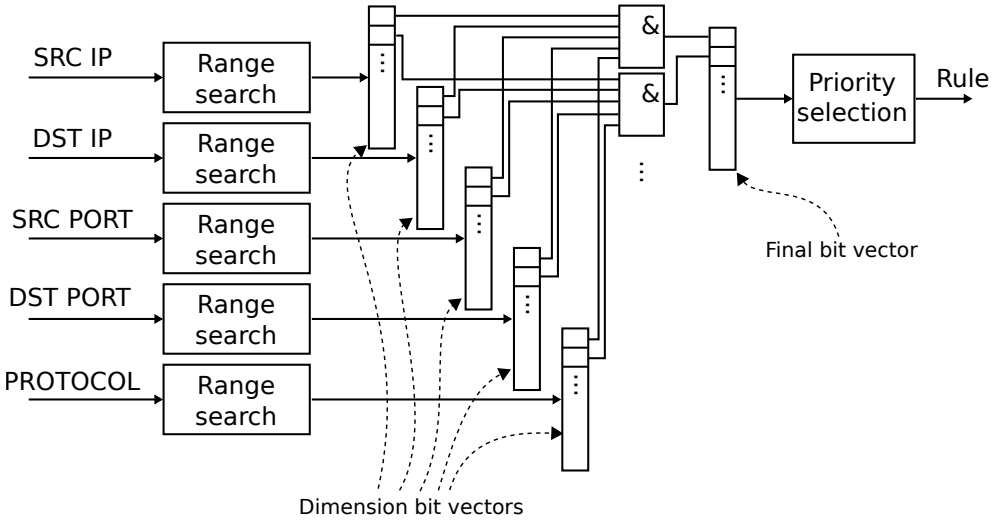


Figure 4.3: Structure of the Bit Vector algorithm.

to the table, several smaller tables are organized recursively. Packet header fields are split into words with suitable width (16 bits, for example). These words are then used as addresses to the tables. Tables are filled with numbers so that packets matching different rules are assigned different numbers (or equivalence classes). Several table outputs are joined by the linear combination and used as an address to the table in next level.

The linear combination is an injective function. For example, combination of values from two tables is computed by multiplying the first value by the number of distinct items in the second table and adding the second value.

The hierarchical structure of tables gradually reduces the number of addressing bits. The last table contains the rules. The scheme of the RFC algorithm is shown in Figure 4.4.

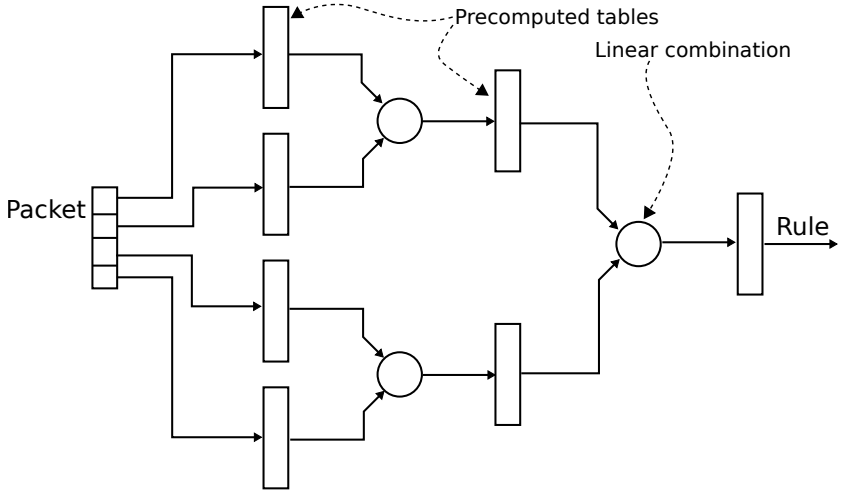


Figure 4.4: Structure of the RFC algorithm.

The RFC algorithm achieves high throughput due to its simple inner operations (only table accesses and linear combinations), but its memory requirements are many times higher than for other algorithms [43].

4.6 Grid-of-Tries

The Grid-of-tries algorithm [46] is optimized for the classification in two dimensions, although it can be extended for more dimensions. Basis of the algorithm is the unibit trie [20]: The algorithm processing one input bit at each tree level (from MSB to LSB) and returning the last valid prefix visited. Grid-of-tries uses the unibit trie to classify in one dimension. The result of the first trie is the pointer to the second trie, where the other dimension is processed. The second trie however contains only subset of all prefixes, because the result of the first trie may disqualify some rules from further searching. The second-level trie attached to prefix p contains only prefixes of the rules which define condition covered by p. This would create $O(n^2)$ memory where n is the number of prefixes. However, the Grid-of-tries algorithm provides optimizations to avoid prefix repetition in the second level tries.

For example, Table 4.1 contains three rules in two dimensions and Figure 4.5 shows how the second level tries are generated (without optimizations). In the case of result $1*$ in the first trie, rule $R3$ certainly does not match the packet, because the corresponding packet header field is 110, 111 or 100, but not 101. The second-level trie therefore does not store prefix 100.

Rule	Dimension 1	Dimension 2	Priority
R1	1*	*	1
R2	1*	00*	2
R3	101	100	3

Table 4.1: Example rules for the Grid-of-tries algorithm.

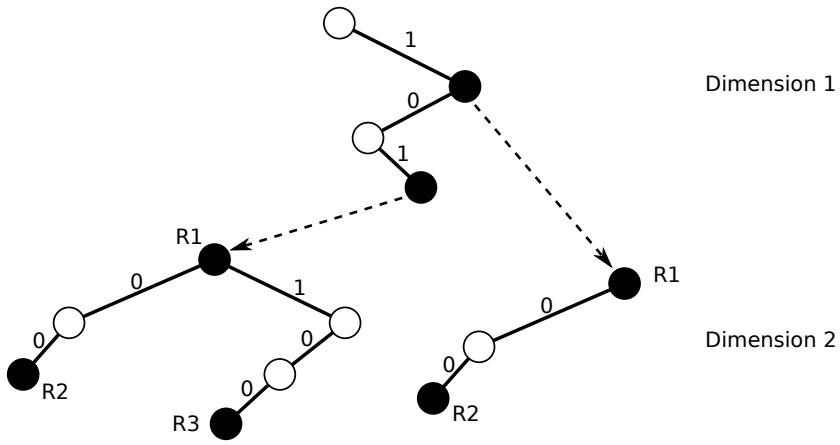


Figure 4.5: Example data structure of the Grid-of-tries algorithm.

4.7 Decomposition methods

The idea of problem decomposition is employed in some of the already described algorithms. Tree based algorithms decompose the problem by dividing the search space into smaller areas. Range matching algorithms decompose the process of packet classification to the

range matching operations and the processing of bit vectors. However, we will use the term *decomposition methods* only for the algorithms described in this section.

Packet classification algorithms described in this section decompose the problem into the sub-problems of finding the longest matching prefix and mapping the prefixes to rules. The example of popular prefix matching algorithm is also presented in the following text.

4.7.1 The Longest Prefix Match Operation

Packet routing in IP networks can be considered as one-dimensional classification – only destination IP address is important for routing. This search on prefixes is the Longest Prefix Match operation as described in Section 3.4, sometimes also called *IP Lookup*. This operation is also important for classification in more than one dimension.

Because the LPM operation is performed in IP packet routing, many approaches were published [20, 45, 32]. The basic algorithm and the data structure for the LPM is the unibit trie. Trie is often modified to process more input bits in each step and to reduce memory requirements. Popular examples of such algorithms are the Tree Bitmap [20] and the Shape Shifting Trie [45]. The LPM operation can be performed very fast: recently published approaches are able to achieve billions of lookups per second [32].

LPM is used in all following packet classification algorithms. It is performed independently in each dimension, as shown in Figure 4.6. Each LPM engine contains all prefixes from one dimension found in the rule set (the prefix set). Because the LPM engines are independent, there is good potential for parallel processing. This thesis does not provide any new LPM algorithm. Some existing approach should be employed instead.

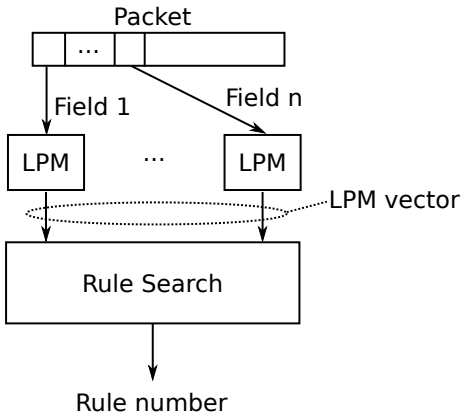


Figure 4.6: Common structure of decomposition packet classification algorithms.

Definition 4.1 (LPM vector)**.** *LPM vector is a vector of results of all LPM engines for one packet. Separate parts of LPM vector can be referenced by the square brackets notation (e.g. LPM[d]).*

4.7.2 Tree Bitmap

The Tree Bitmap algorithm [20] is presented as an example of popular and advanced LPM algorithm. It is based on the basic trie, but processes more input bits at each tree level. Number of bits processed at once is called *stride*. Direct extension of trie would create a tree structure where each node represents a small trie with the depth *stride*. This would

however require to store rather large data structure per node: One pointer for each possible resulting prefix (up to $2^{stride} - 1$ pointers), and one pointer for each of the following nodes (up to 2^{stride} pointers).

Tree Bitmap algorithm avoids storing so many child pointers by placing all sibling nodes consecutively in the memory. Then only one pointer to the first child node and the bitmap of valid child nodes needs to be stored in the node. Pointers to the other child nodes can be easily computed by adding the number of preceding ones in the bitmap to the first child pointer.

Similar approach is used for the prefixes. All prefixes for one node are stored consecutively in the memory and the node stores only pointer to the first prefix and a bitmap of valid prefixes (provided that some ordering of prefixes is defined in the node).

Figure 4.7 shows the example of one Tree Bitmap node connected in the tree, together with the placement of child nodes and prefixes in the relevant tables.

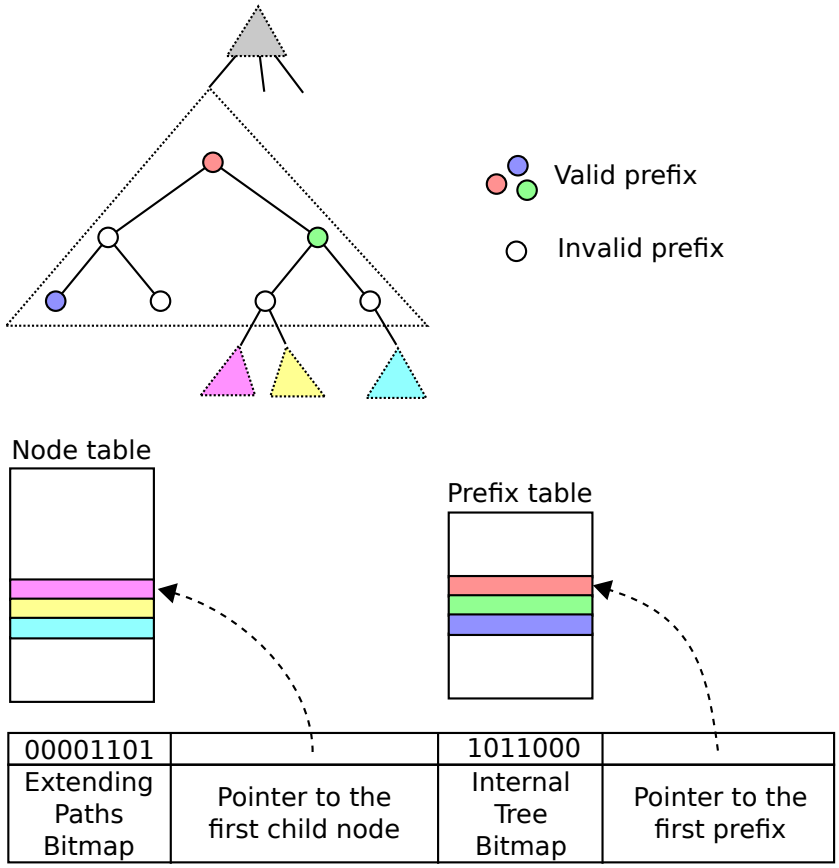


Figure 4.7: Example of one Tree Bitmap node for $stride = 3$.

The Tree Bitmap algorithm was improved by the Shape Shifting Trie algorithm [45] (SST). This algorithm uses nodes which may have different shape than the full (symmetric) trie. This property is especially useful when the tree segment has the shape of long path without branching. In that case, SST uses nodes with the corresponding shape and processes more input bits in each node.

4.7.3 Direct Cartesian Product

Direct Cartesian product algorithm [46] precomputes a Cartesian product table which contains correct rule numbers for all possible LPM vectors. After the LPM is performed for all dimensions, the LPM vector is searched for in the Cartesian product table (possibly implemented as a hash table). Because of the multiplicative nature of the Cartesian product, this table may become extremely large. For example, Cartesian product table for the smallest rule set from Table 5.1 (rules1 – 103 rules) would contain 1 290 240 items.

4.7.4 DCFL

The basic algorithm was improved in 2005 by the Distributed Crossproducing of Field Labels [47]. LPM is modified to return all valid prefixes (not only the longest one) for the given field value. Valid prefixes are then filtered by the aggregation network of small set membership query filters. Inputs of each filter are two sets of prefixes (or labels, in general). The filter then performs a set membership query for each possible pair (Cartesian product) of labels. The result of the filter is another set of labels. The result of the last filter is in fact a set of rules, from which the one with the highest priority is selected. Figure 4.8 shows the algorithm structure for classification in four dimensions.

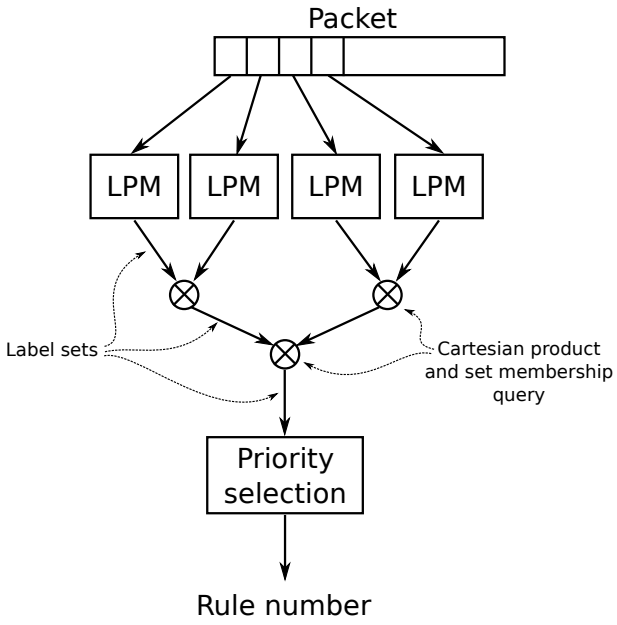


Figure 4.8: Structure of the DCFL algorithm.

Even when Cartesian products are generated in a distributed way, they are still a weak point of the algorithm, because Cartesian product is multiplicative in nature. If, for example, both input sets of the set membership query filter have 5 items, then the filter has to perform $5 \times 5 = 25$ set membership queries.

4.7.5 MSCA

Multi Subset Crossproduct Algorithm [17] brought major improvements to decomposition methods in 2006. In this work, Dharmapurikar et al. replace Cartesian products by *pseudorules* (described in detail in Section 4.7.6 of this work). Because pseudorules expansion is

still similar to Cartesian product, authors provide heuristics how to break the rule set into several subsets and eliminate the majority of pseudorules. The LPM operation is slightly modified to return a result for each subset, because subsets may contain different prefix sets. A Bloom filter ([8], see Section 4.7.7) is associated with each subset to perform set membership query. If the Bloom filter output is true, one rule table memory access is performed to retrieve the resulting rule or pseudorule. Figure 4.9 shows the basic algorithm structure.

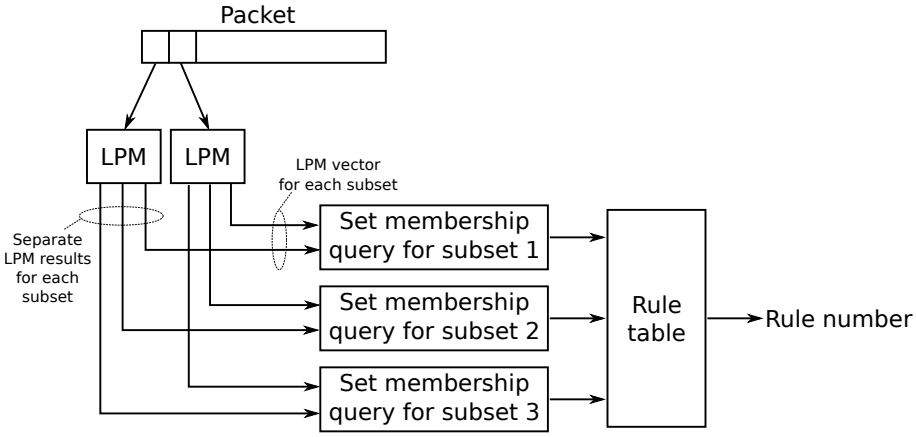


Figure 4.9: Structure of the MSCA algorithm for two dimensions and three subsets.

MSCA also identifies rules that generate excessive amount of pseudorules. These rules are called *spoilers* and are treated in a separate branch of the algorithm to further reduce number of pseudorules. In hardware implementation, spoilers are moved to a small on-chip TCAM.

MSCA suffers from placement of the rule table in the external memory. Wide data word (over 100 bits if IPv6 is not used) is needed to fetch the rule in one access.

4.7.6 Pseudorules

The concept of pseudorules was introduced in MSCA and is used also in all four algorithms that follow in this work. First we describe in detail how pseudorules are created: Pseudorules must be added to the rule set to cover all valid combinations of LPM results. In fact, a pseudorule is always a special case of some rule. We explain the emergence of pseudorules on the example in Figures 4.10 and 4.11 and Table 4.2. We can see a simplified classification in two three-bit dimensions with three rules. In each dimension, unibit trie is shown to illustrate the LPM operation. Colored arcs are the rules.

For example, LPM vector for packet with header fields $(111, 100)$ is $(1*, 100)$. This combination is not in the original rule set, but it is clear that the correct result is rule $R1(1*, *)$. Therefore, pseudorule $P1(1*, 100)$ must be added to handle this situation. Table 4.2 contains all rules and pseudorules together. The *target rule* in this table points to the correct classification result.

The generation of pseudorules is similar to Cartesian product, and may potentially expand the rule set significantly, but not all possible combinations of prefixes need to be added. Prefix combinations matching no rule are not pseudorules. If the universal rule is in the rule set, then all possible combinations must be added, because all of them match some rule (at least the universal rule). However, this rule can be removed from the rule set and

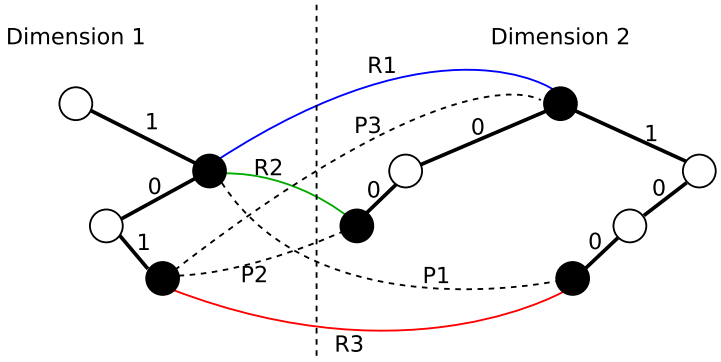


Figure 4.10: Three rules $R1, R2, R3$ and three added pseudorules.

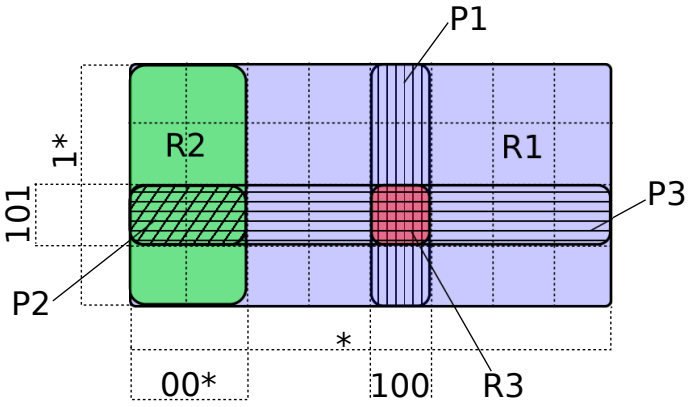


Figure 4.11: Graphical representation of rules $R1, R2, R3$ and pseudorules $P1, P2, P3$.

Rule	Dimension 1	Dimension 2	Priority	Target rule
R1	1*	*	1	
R2	1*	00*	2	
R3	101	100	3	
P1	1*	100	1	R1
P2	101	00*	2	R2
P3	101	*	1	R1

Table 4.2: Rules and pseudorules.

returned as a result only if no other rule matches the packet. Therefore, pseudorules form a subset of Cartesian product of all prefix sets. The algorithm generating all pseudorules from the rule set is shown in Algorithm 4.1. The algorithm creates pseudorules by finding prefix combinations that match some rule.

Packet classification algorithms deal with pseudorules in the Rule Search Stage (see Figure 4.6). The problem of finding the correct rule has the interesting property of mapping huge number of inputs (LPM vectors) to only moderate number of outputs (rule numbers).

Algorithm 4.1 Generating pseudorules from the rule set.

Input: Rule set R without the universal rule and with range conditions converted to prefixes.

1: **for all** dimension d **do**
2: Create empty prefix set S_d.
3: **for all** rules $r \in R$ **do**
4: Add $r.d$ to S_d.
5: **end for**
6: $\{S_d$ is now complete prefix set for dimension $d.\}$
7: **end for**
8: Create empty set of pseudorules P.
9: **for all** rules $r \in R$ sorted from the highest to the lowest priority **do**
10: **for all** dimensions d **do**
11: Create reduced prefix set SR_d by selecting prefixes from S_d which are covered by $r.d$.
12: **end for**
13: Create set CP as the Cartesian product of all reduced prefix sets SR_x.
14: **for all** candidate pseudorules $cp \in CP$ **do**
15: **if** $cp \notin P$ **then**
16: Add cp to P with the same priority as r and the target rule pointer set to r.
17: **end if**
18: **end for**
19: **end for**

Output: P

4.7.7 Bloom Filters

Bloom filter [8] is a data structure that is used to test whether an element is a member of a set. Its greatest advantage is the fact that the elements themselves are not stored in the data structure, therefore the Bloom filter is very space-efficient. The data structure is however probabilistic, with the possibility of false positives. The element which was not added to the Bloom filter may be incorrectly reported as present.

The Bloom filter stores an array of m bits. When the empty Bloom filter is created, all m bits are set to 0. There are also k different hash functions, each of which maps the elements to one of m bits with the uniform random distribution. Bloom filter supports two operations:

- *Add an element*: All k hash functions compute the hash of the element being added to get k bit positions. All k bits of the bit array are set to 1.
- *Set membership query*: All k hash functions compute the hash of the element being queried to get k bit positions. If all k bits of the bit array are set to 1, then the queried element is reported as present.

Figure 4.12 shows the steps performed in both operations. Adding an element sets all selected bits to 1, query operation checks whether all selected bits are set to 1.

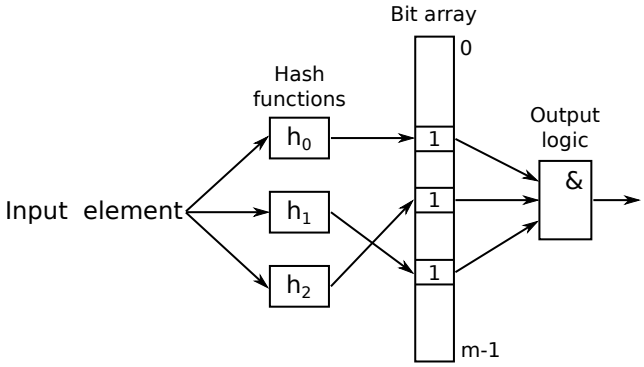


Figure 4.12: Example of Bloom filter for $k = 3$.

The probability of false positive is obtained by the following consideration: The probability that a certain bit of the bit array is *not* set by a single hash function during the insertion of a single element is

$$1 - \frac{1}{m}$$

The probability that the bit is not set by any of k functions is

$$\left(1 - \frac{1}{m}\right)^k$$

and the probability that the bit is not set after n elements is inserted is

$$\left(1 - \frac{1}{m}\right)^{kn}$$

The probability that the bit is set to 1 is therefore

$$1 - \left(1 - \frac{1}{m}\right)^{kn}$$

The element is erroneously reported as present in the set if all k bits are set to 1:

$$\left(1 - \left(1 - \frac{1}{m}\right)^{kn}\right)^{k}$$

After some simplification of this formula and further calculations it may be observed that the optimal k for the given m and n is approximately $0.7\frac{m}{n}$. In that case the probability of false positive is $0.6185^{\frac{m}{n}}$.

Bloom filters are used in the MSCA algorithm as well as in many other algorithms and applications [16, 29, 15]. Although the Bloom filter was not designed for that, it can be particularly effective in the hardware implementation [10].

Chapter 5

Analysis of Packet Classification Algorithms

All the algorithms presented in the previous chapter have the speed – memory trade-off. Faster algorithms require large memory to store the data structures, while more memory-saving algorithms are not capable to achieve high throughput.

The target of this thesis is to present the new algorithm with the throughput of 100 Gb/s. For such high speed requirement, there are certain limitations on the algorithm design. From the presented approaches, the decomposition methods have the greatest potential to achieve high throughput, because they may be designed to perform the fixed number of steps for each packet [46]. Another technique that is very suitable when designing high-speed devices is the pipelining. The designed algorithm should therefore have the shape of high-speed pipeline with fixed number of steps, without loopbacks or cycles.

The Direct Cartesian product algorithm [46] employs only the hash table in the Rule Search stage and thus achieves the constant time of this step. This speed is however bought at the cost of storing all possible LPM vectors, which makes the memory requirements too high. If the hash table size is too high, then DRAMs must be used for its implementation, which brings significant slowdown compared to SRAMs.

MSCA [17], on the other hand, replaces LPM vectors by pseudorules and reduces their number by two techniques – spoilers removal and division of rule set into subsets. Memory consumption of MSCA is much better, the rule table fits into single SRAM. However, the whole rule must be read from the memory to classify one packet, which brings high requirements on the external memory throughput.

From the stated facts we conclude that:

- The designed algorithm must decompose the problem of packet classification into easier tasks.
- The designed algorithm must be convertible into pipelined hardware representation.
- The data structures of the algorithm must fit into single SRAM.

In this chapter we analyze the pseudorules in several rule sets and study the impact of the spoilers removal technique in detail, because the proper combination and improvement of the Direct Cartesian product and MSCA algorithms is a promising path to find a new algorithm suitable for 100 Gb/s networks.

5.1 Rule Sets

Six rule sets are used for experiments in this thesis. Four of them are real-life rule sets from university network firewalls (rules1-4), two are synthetic rule sets generated by ClassBench [48] tool (synth1-2). Numbers of rules (before and after expansion of ranges to prefixes) and numbers of unique prefixes in each dimension are shown in Table. 5.1.

Rule set	Rules (original)	Rules (no ranges)	Source IPs	Dest. IPs	Protocols	Source Ports	Dest. Ports
synth1	219	374	55	53	1	14	1
synth2	394	649	65	57	1	14	1
rules1	103	109	28	48	4	6	40
rules2	173	184	84	84	3	1	16
rules3	275	275	46	64	3	1	22
rules4	1 107	1 244	158	80	4	1	56

Table 5.1: Basic properties of rule sets.

It is worth noting that numbers of individual prefixes are significantly smaller than the number of rules. This fact was observed by other researchers [47, 17] and is also visible in Table 5.1. It is more evident for larger rule sets (synth2, rules4). This supports the idea of problem decomposition: Independent LPM searches should be easy, because there are small numbers of prefixes in separate dimensions.

5.2 Pseudorules Analysis

Dharmapurikar et al. in [17] observes that the number of pseudorules is up to 200 times higher than the number of original rules. Experiments with Algorithm 4.1 however show even higher numbers in some cases. Figure 5.1 shows the histogram of pseudorules for one rule set giving the number of associated pseudorules for each rule. It can be seen that the majority of pseudorules is generated by minority of rules (note the logarithmic scale of the vertical axis). In this particular case, the top 10 rules (10 % from a total of 103 rules) generate 42.34 % pseudorules. Removing those rules should decrease the number of pseudorules significantly. Histograms for other rule sets show the similar feature.

Table 5.2 and Figure 5.2 show the number of pseudorules for the test rule sets and for different numbers of removed spoilers. The size of Cartesian product CP in line 13 of Algorithm 4.1 is used to identify rules generating most of pseudorules. Detailed analysis and other options for spoilers identification are given in [26].

As can be seen from the second column of Table 5.2, number of pseudorules is up to 10 049 times more (!) than the number of original rules (rules1). It can also be seen from the following columns that removing spoilers significantly reduces the number of pseudorules for each testing rule set. After removing of 32 spoilers, only 6 (rules1) to 36 (rules2) percent of pseudorules remain.

There are two beneficial effect of the spoilers removal that affect the number of pseudorules:

- The *direct* effect is clear from the pseudorules distribution shown in Figure 5.1. Majority of pseudorules is generated by minority of rules. By removing only a small

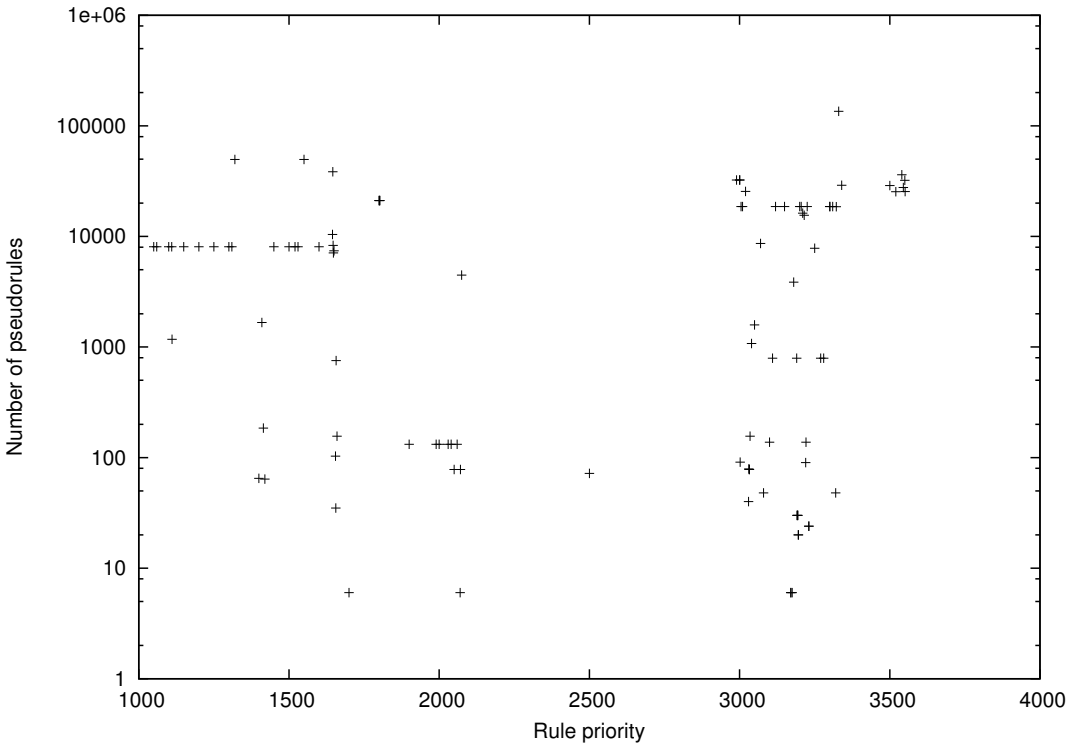


Figure 5.1: Pseudorules histogram in the rule set rules1.

amount of the worst rules, many pseudorules are avoided.

- The *indirect* effect rises from the nature of Cartesian product: By removing spoilers also the number of unique prefixes may be lowered and thus the Cartesian products are smaller.

Figure 5.3 shows the pseudorules histogram after removing pseudorules: The graph is similar to the previous graph in Figure 5.2, but the highest points of the graph are missing – these rules are identified as spoilers and removed. Figure 5.4 contains data from both Figures 5.1 and 5.3 with data points sorted by their values. It can be seen that also the rules that are not removed now generate less pseudorules. The reason for this is the indirect effect of spoilers removal: Removing spoilers removes also some unique prefixes and thus produces smaller Cartesian products.

5.3 Conclusions

It is clear from Table 5.2 that storing all pseudorules would require excessive amount of memory, which makes the Direct Cartesian product algorithm unusable in practice. The MSCA, which requires significantly less memory, employs more complex Rule Search stage which results in non-deterministic throughput. While these two algorithms are probably the best candidates for the 100 Gb/s solution, none of them meets the requirements imposed by such high-speed network.

The following chapter presents the algorithm that requires significantly less memory than the Direct Cartesian product algorithm and is significantly faster than MSCA.

Rule set	Pseudorules after spoilers removal				
	0	4	8	16	32
synth1	40 810	17 267	14 187	9 879	4 786
synth2	51 870	24 281	21 882	18 762	11 848
rules1	1 105 392	724 046	601 512	250 366	74 974
rules2	338 200	317 020	167 120	151 472	122 480
rules3	31 283	21 731	17 390	8 758	2 898
rules4	112 906	41 226	22 295	19 584	16 937

Table 5.2: Numbers of pseudorules after removal of 0, 4, 8, 16, 32 spoilers.

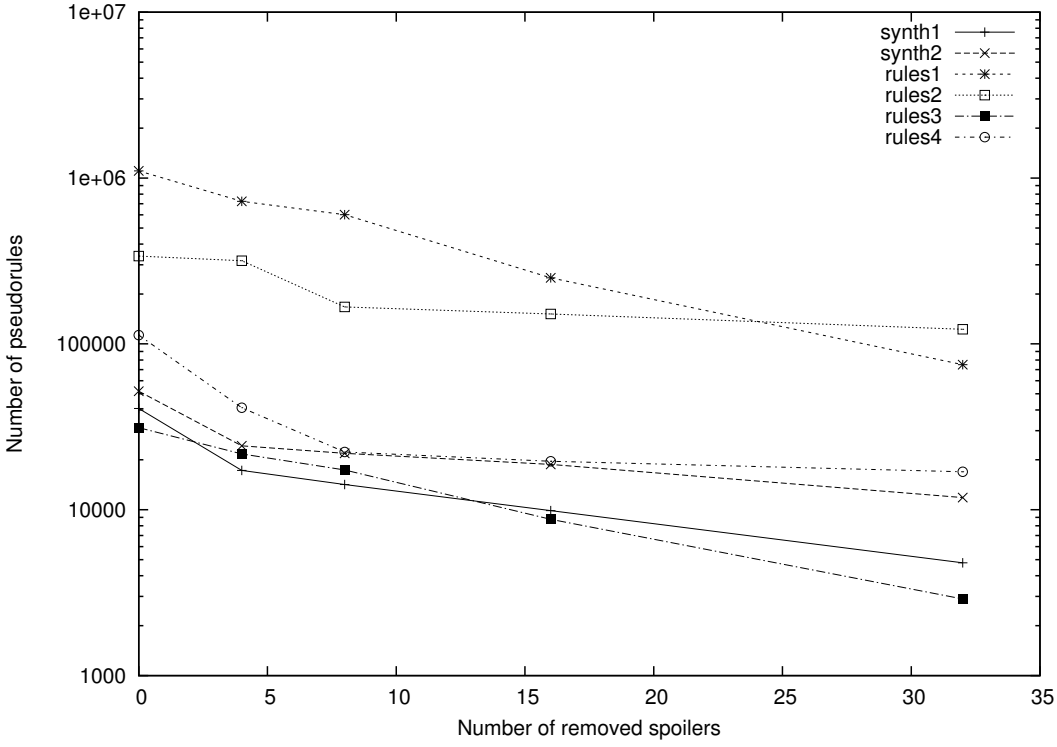


Figure 5.2: Pseudorules decrease after removing spoilers.

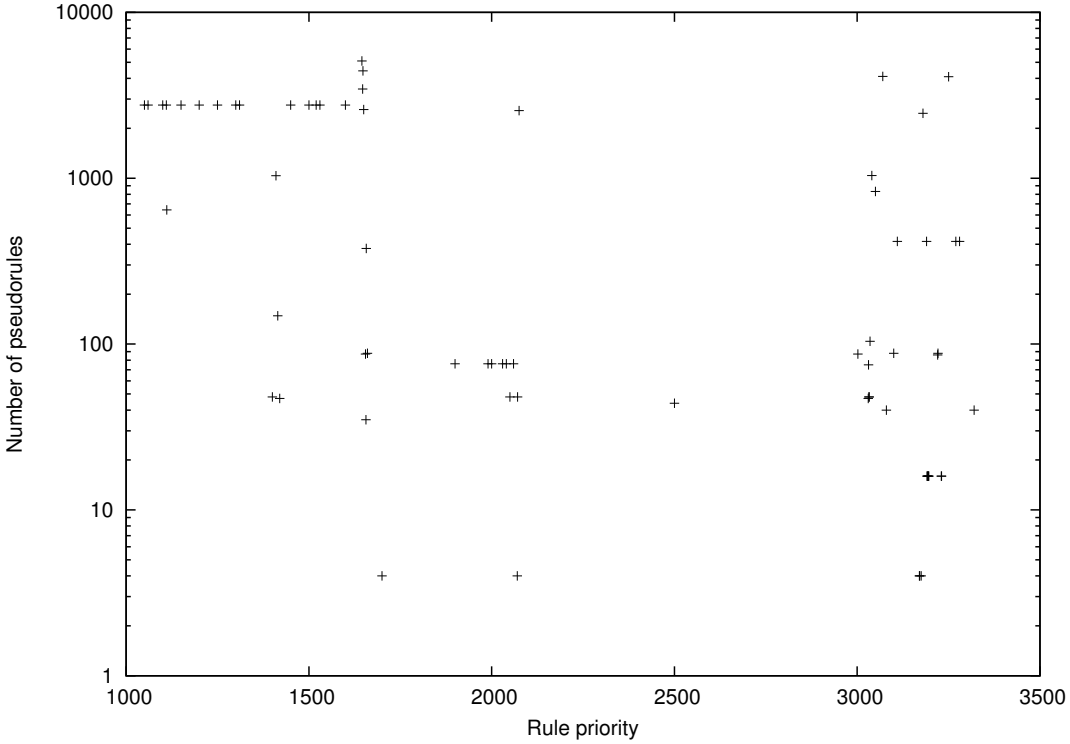


Figure 5.3: Pseudorules histogram in the rule set rules1 after removing 32 spoilers.

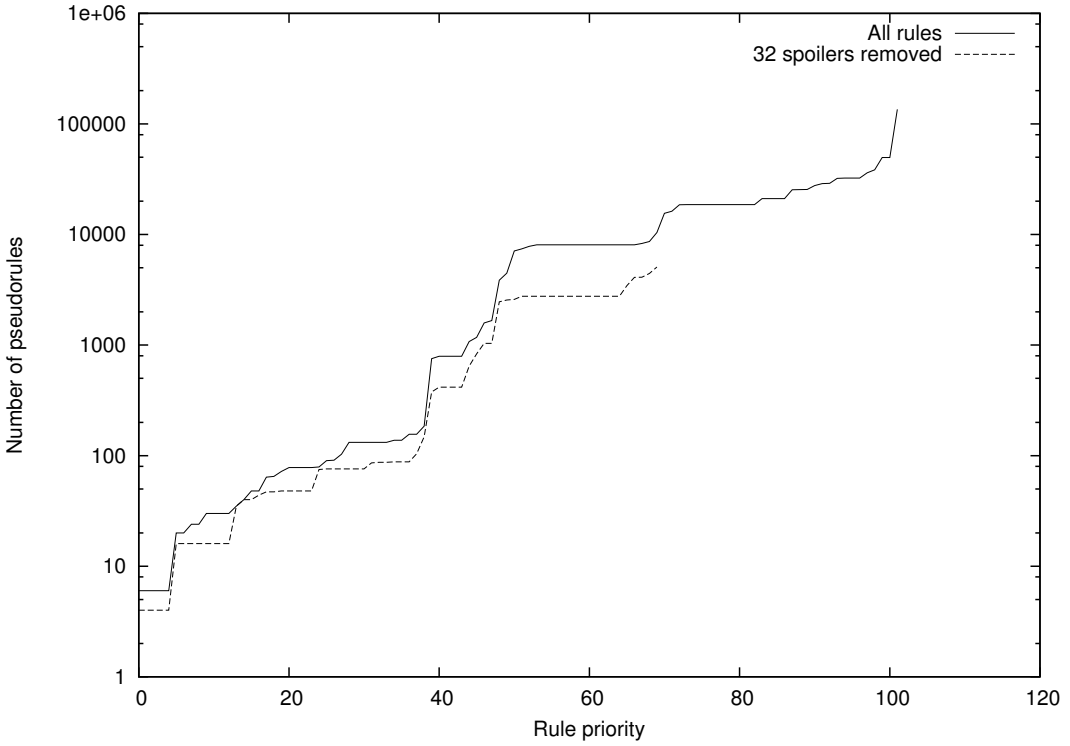


Figure 5.4: Comparison of pseudorules before and after spoilers removal. Data points are sorted by value.

Chapter 6

Perfect Hashing Crossproduct Algorithm

This chapter presents the Perfect Hashing Crossproduct Algorithm (PHCA, [41]). The algorithm is inspired mostly by the MSCA, as it also employs the concept of pseudorules. It differs, however, in the technique used to map LPM results to the correct rule number. While the Multi Subset Crossproduct Algorithm focuses on lowering number of pseudorules by removing spoilers and dividing the rule set into subsets, the Perfect Hashing Crossproduct Algorithm takes another approach.

PHCA meets all the goals stated in the previous section: It decomposes the problem of packet classification into easier tasks, it is directly represented as a pipeline of processing elements, and its largest data structure fits into single SRAM.

6.1 Introduction

Chapter 4 has introduced several algorithms for packet classification. The family of decomposition algorithms is probably the most mature and some of the most promising algorithms, such as MSCA, belong to it. However, Chapter 5 explains why none of the described algorithms meets the requirements imposed by the 100 Gb/s network.

The algorithm introduced in this section addresses the most critical part of the decomposition algorithms, which is the Rule Search stage. The problem in the Rule Search stage is that it has to handle large amount of possible inputs (all possible LPM vectors), while some inputs match no rule, some match one rule and some match several rules, from which the one with the highest priority has to be selected. The 100 Gb/s target requires that this processing is done very fast.

One candidate for implementation of the Rule Search stage is the ordinary hash function. This well known lookup algorithm finds the result in constant time if there is no collision in the hash table (more inputs hashed to one memory location). However, collision handling complicates the processing. Moreover, the hash table requires all of its items to be stored, because they must be compared to the input. Therefore, the ordinary hash function does not suit well for the implementation of the Rule Search stage.

We discuss the possibilities to remove hash function collisions. Hash functions that avoid collisions are called *perfect hash functions*. These are divided into two groups:

- *Static* perfect hash functions do not support runtime inserting nor deleting of the keys. Complete set of keys must be known prior building the function. Once the

function is built, only the lookup operation is supported [13, 12, 25]

- *Dynamic* perfect hash functions support runtime modification of the key set. They are essentially more complex than static perfect hash functions [18, 19, 38].

6.2 Use of Perfect Hashing in Packet Classification

We start the algorithm description with the simplified situation when there are no pseudorules. We add pseudorules handling later in the text.

In the domain of packet classification, the general concept of hash table specializes to the rule table and the set of hash keys specializes to the set of all rules expressed as LPM vectors. This set is known in advance, therefore the static perfect hash function suffices in this case. The result of the hash function is the pointer to the rule table.

The perfect hash table (rule table) stores all rules, except for the universal rule, which covers all packets. The reason for removing the universal rule is explained later. The perfect hash function maps each LPM vector to the correct rule if the packet matches some rule. Therefore, the classification algorithm performs the LPM and then computes the perfect hash function. The result is a pointer into the rule table, where the rule is read from. This rule is then compared to the original packet. If it matches, then the correct rule was found. If not, then the packet matches no rule or the universal rule (if present). Figure 6.1 shows the basic structure of PHCA.

This arrangement allows the perfect hash function to return any (even incorrect) result if the packet matches no rule. This situation may occur only if the LPM vector does not correspond to any rule and is always rectified by matching the packet to the rule in the rule table. In the case of no match, the universal rule is returned. Now it is also visible why the universal rule is not included in the perfect hash function construction: If it was included, then all LPM vectors not corresponding to any rule would have to be correctly hashed to the universal rule. Allowing the perfect hash function to return in some cases an incorrect rule number removes the necessity to handle the complete Cartesian product of all prefix sets and therefore saves significant amount of memory in the perfect hash function implementation.

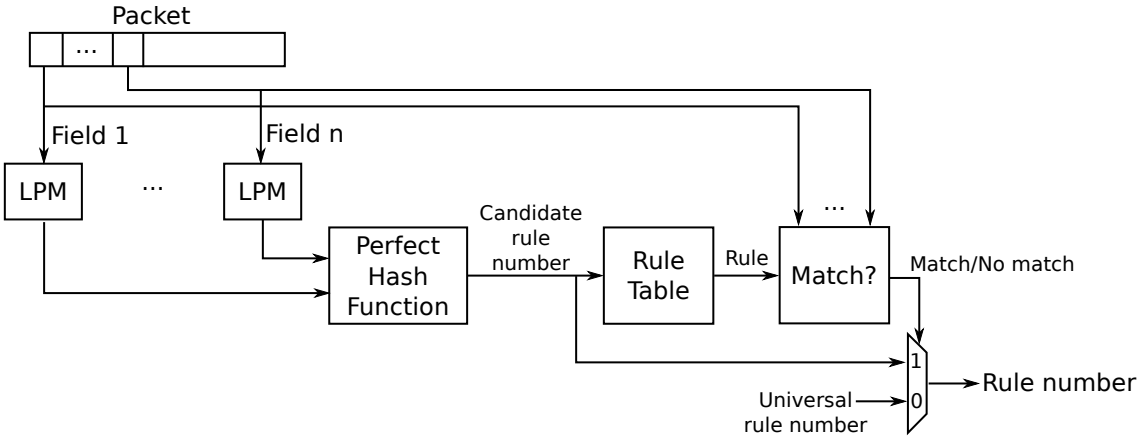


Figure 6.1: Basic structure of the PHCA.

We continue by describing the perfect hash function construction algorithm as presented by Czech et al. in [13]. The perfect hash construction algorithm creates an acyclic graph

where edges are the hash keys and vertices are results of two different ordinary hash functions. Vertices are then assigned values so that they sum up to the desired hash value. The hash construction is shown in Algorithm 6.1. The function associations (a, b) are associations between rules in the LPM vector representation and the rule numbers, which are also pointers to the rule table. Prefixes in LPM vectors are represented by some symbols, for example 16-bit integers. The created perfect hash function has the same number of inputs and outputs and is bijective.

Algorithm 6.1 Construction of the hash function.

Input: Set of function associations A, each association is a tuple (a, b) of the LPM vector and the correct rule number.

1: Create new graph G with $g = c|A|$ vertices and no edges, where $c > 1$. The real number c is used to increase the graph size when needed.
2: Pick two different ordinary hash functions f_1, f_2 that output integers from interval $[0, g - 1]$.
3: **for all** $(a, b) \in A$ **do**
4: $h_1 \leftarrow f_1(a)$
5: $h_2 \leftarrow f_2(a)$
6: Add an edge between vertices h_1 and h_2 into the graph G and label that edge b.
7: **end for**
8: **if** G contains cycle **then**
9: Increase c and repeat the algorithm. The increment is typically a small number, for example 0.2.
10: **end if**
11: Associate values to each vertex such that for each edge the sum of the values of both its vertices is the value of that edge. This can be done by depth-first search algorithm, because the graph is acyclic.

Output: f_1, f_2 and vertex values of G.

After the graph is created, the hash value computation is simple. At first, two ordinary hash functions f_1 and f_2 of the input LPM vector are computed. Then two vertex values are read from the Vertex Table and added. For each vertex, only one integer is stored. Functions f_1 and f_2 may be virtually any convenient hash functions, the only requirement is that the same functions are used in the perfect hash construction and the following computation. CRC32 is used for experiments in this thesis.

Complete table of rules is stored in PHCA, because if packet matches no rule, the perfect hash function still returns some value (the function is not built for LPM vectors which are not rules). Therefore, packet must be compared to the selected rule: If it matches, the correct rule was found. Otherwise, the packet matches no rule or universal rule (if present). Structure of the Perfect Hashing Crossproduct Algorithm is shown in Figure 6.2.

6.3 Pseudorules in PHCA

The arrangement described so far does not support pseudorules. To add the pseudorules support to the algorithm, surprisingly little must be done. Refer to the Algorithm 4.1: each pseudorule has the number of correct rule associated to it. It is therefore known which rule is the desired algorithm output for each pseudorule. This information is used to create additional input to the perfect hash function construction Algorithm 6.1. In addition to

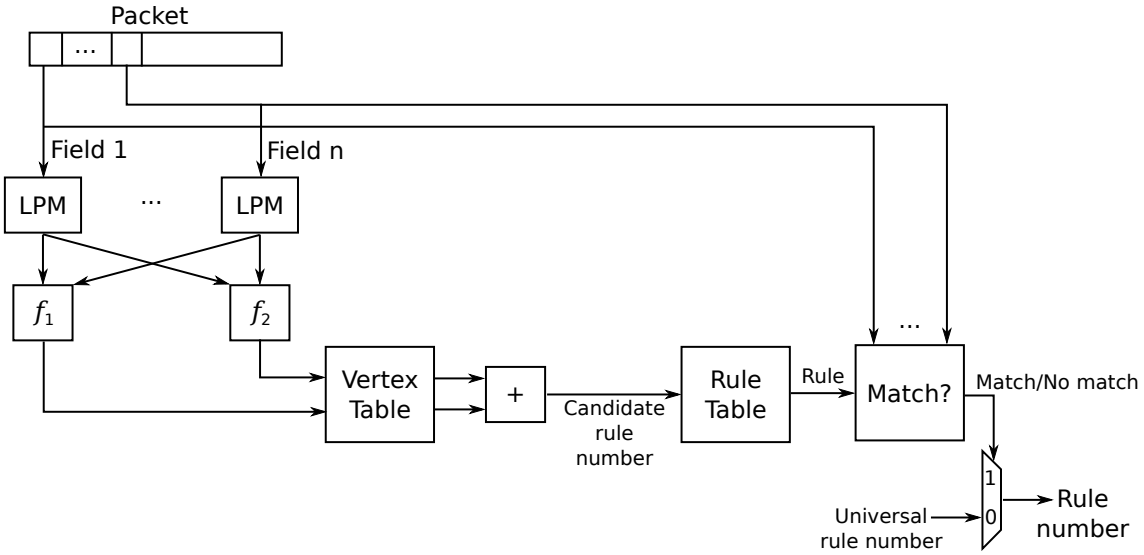


Figure 6.2: Detailed structure of the PHCA.

the function associations (a, b) between the rules in the LPM vector representation and the rule numbers, the input A now contains also the associations between the *pseudorules* in the LPM vector representation and the *rule* numbers.

The perfect hash function is not bijective anymore. It maps large number of pseudorules into the smaller number of rules. In fact, the hash function is not perfect anymore. It instead contains many collisions among each rule and all of its associated pseudorules. The use of intended hash function collisions is an innovative, non-traditional application of perfect hash functions. The important point is that none of pseudorules is stored in PHCA – the rule table still contains only rules after adding the pseudorules support into the algorithm. Therefore, a significant amount of memory is saved.

6.4 Perfect Hash Example

Table 6.1 shows rules and pseudorules a from Table 4.2 together with the desired perfect hash function output b – the correct rule number. It also shows two hypothetical hash functions f_1, f_2 for LPM vectors. The functions are completely made up only for the illustration. Figure 6.3 then shows the acyclic graph created for $g = 8$ (therefore c is $\frac{4}{3}$). Vertices are represented by circles with their names (addresses) outside them. Vertex values are the numbers inside the circles. Graph edges are shown as straight lines with their names near them.

At the beginning, the graph contains only vertices, but no edges. Graph edges are then created by connecting the edges according to the pseudorules (lines from Table 6.1). Because the graph is acyclic, vertices can be associated values by a depth-first search algorithm. Is this case, vertices 0 and 4 are selected as starting points for the numbering. Zero values are assigned to them, while the remaining vertices are numbered so that the sum of each edge vertices equals b from the Table 6.1.

Figure 6.4 shows the vertex table and the example of how the hash function is computed.

Rule	LPM vector a	Maps to b	$f_1(a)$	$f_2(a)$
R1	(1*, *)	1	0	7
R2	(1*, 00*)	2	6	0
R3	(101, 100)	3	5	4
P1	(1*, 100)	1	0	4
P2	(101, 00*)	2	1	3
P3	(101, *)	1	3	2

Table 6.1: Two hash functions' results for inputs in the form of LPM vectors

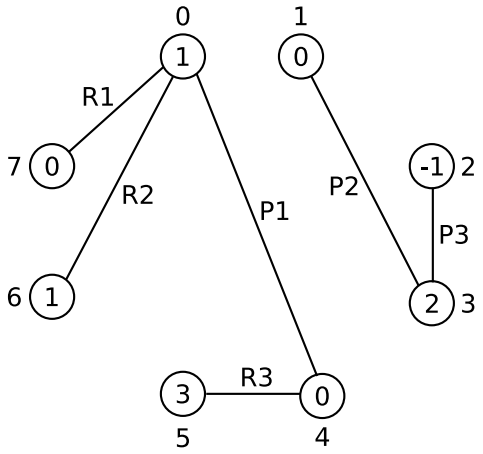


Figure 6.3: Example graph with 6 rules and 8 vertices.

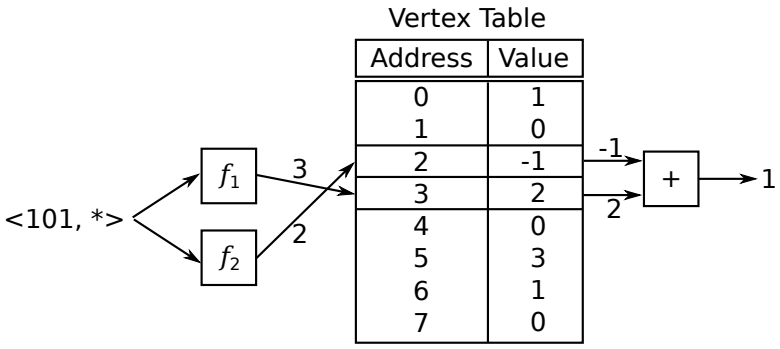


Figure 6.4: Example of computing the hash function.

6.5 Effective TCAM implementation in FPGA

Based on the good results in Chapter 5, the spoilers removal technique is also used in PHCA. The memory for spoilers directly stores fixed number of rules. Eight spoilers are removed in all experiments unless stated otherwise. Spoilers are easily matched by a TCAM.

The example of implementation of small CAM in FPGA is given in [9]. FPGA look-up tables (LUTs) are filled with a bit pattern such that only the correct (matched) input bits generate logical 1 at the LUT output. The LUT truth table has only one logical 1 in the result column. Logical AND over all LUTs then computes the match for the given CAM row. The idea of using look-up tables (LUTs) to directly match input data can be directly

extended to support 6-input LUTs used in current FPGAs. The TCAM functionality (storing *don't care* bits) can be easily added by adding more logical ones in the LUT truth table. Each don't care bit doubles the number of ones in the LUT truth table, LUT storing all don't cares always returns logical one. Figure 6.5 shows the basic scheme for one TCAM row.

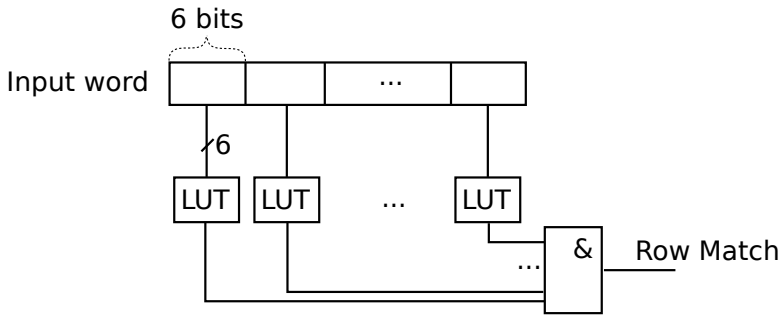


Figure 6.5: Scheme of one TCAM row in FPGA.

In the case of classification rules, one TCAM word is $128 + 128 + 8 + 16 + 16 = 296$ bits wide (source IPv6, destination IPv6, protocol, source port, destination port). Given the fact that one LUT matches 6 input bits, one TCAM word uses 50 LUTs and the whole memory of 8 spoilers uses 400 LUTs to store data.

This basic scheme stores 6 bits per LUT and is capable of matching one input word in each cycle. However, if the slower operation is acceptable, then the TCAM can be implemented more effectively. We can split the input of each LUT into two parts: n LUT input bits are the row index, because each LUT now matches part of 2^n rows sequentially. $6 - n$ LUT input bits are the part of currently matched word. Therefore, each LUT now stores $2^n(6-n)$ bits. It also takes 2^n cycles to match single input word. Figure 6.6 shows the improved scheme of TCAM. The basic TCAM scheme corresponds to the improved scheme for $n = 0$. Table 6.2 shows how different settings of n affect TCAM size and throughput. The effectivity takes into account also bits wasted at the end of word, if the word width (296 bits) is not divisible by $6 - n$. The maximum effectivity is reached at $n = 4$, where matching takes 16 cycles, and each LUT stores 32 bits (2 bits for each of 16 matched rows). This option is not shown in Table 6.2, because we consider only TCAMs with 8 rows.

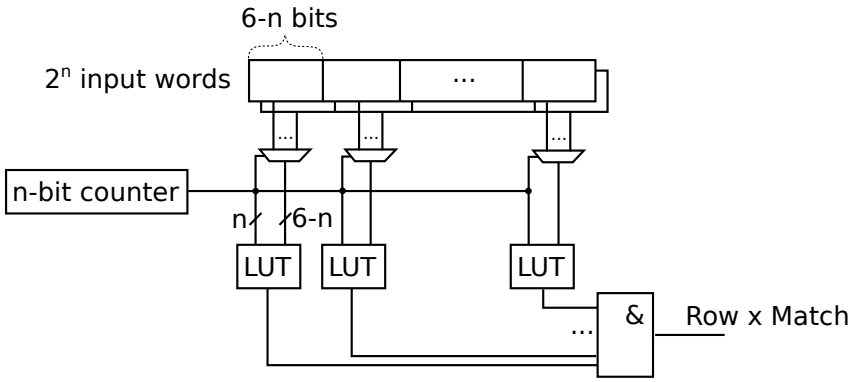


Figure 6.6: Scheme of optimized TCAM in FPGA.

n	LUTs	Match time (cycles)	Effectivity (bits/LUT)
0	400	1	5.92
1	240	2	9.87
2	148	4	16.00
3	99	8	23.92

Table 6.2: Different settings of optimized TCAM storing 8 rows of 296 bits.

6.6 PHCA Evaluation

The algorithm addresses the issue of large number of pseudorules differently than methods mentioned in Chapter 4. Instead of lowering their number, pseudorules are not stored in the algorithm. However, the number of pseudorules affects the size of data structure of the hash function – size g of the Vertex Table must be higher than the number of pseudorules. This means that this algorithm may be significantly improved if the number of pseudorules (domain of the perfect hash function) is reduced.

6.6.1 Memory

PHCA contains four main memories: LPM engines, the Vertex Table, the Rule Table and the spoilers. Memory of separate LPM engines is evaluated in Chapter 10, because most discussed algorithms share this step. The spoilers are stored in TCAM, as explained in the previous section.

Rule Table size is proportional to the size of rule set. The size of one rule is discussed here: To store the IP address condition, the IP address value and prefix length needs to be stored. 128 bits for IP address can accommodate both IPv4 and IPv6, higher 96 bits remain unused for IPv4. The IPv6 prefix length needs 8 bits to store 129 different values (0 to 128). Again the IPv4 prefix length can be stored in the same place in memory, only 2 higher bits remain unused. Protocol and port conditions are stored as ranges. Table 6.3 sums up all conditions that make up the entry of the Rule Table. Table 6.5 shows the size of the Rule Table for different rule sets.

The Vertex Table is expected to be the largest memory in PHCA. Its size is approximately proportional to the number of pseudorules. More precisely: Number of items in the Vertex Table size depends on the number of pseudorules and the constant c from Algorithm 6.1. The sooner acyclic graph is found, the smaller c can be. The lowest possible number of vertices in acyclic graph is $e + 1$ where e is the number of edges. In our case, e is the number of pseudorules. However, probability of creating such random graph is extremely low. Theoretical results in [13] show that the probability of creating random acyclic graph approaches nonzero constant for $c > 2$. In most experiments the acyclic graph is found for c between 2.1 and 2.45, which supports the theory. This is shown in Table 6.4.

The smallest possible size of one vertex in the Vertex Table is $\log_2$(rule set size − spoilers). In practice however, one fixed size for all rule sets is more feasible for implementation independent on the rule set. 16 bits are chosen as a balance between the item size and the maximal number of rules supported. Table 6.5 shows the size of the Vertex Table.

Item	Length
Source IP address	128
Source IP prefix length	8
Destination IP address	128
Destination IP prefix length	8
Protocol min	8
Protocol max	8
Source port min	16
Source port max	16
Destination port min	16
Destination port max	16
Total	352

Table 6.3: Storage size of one rule.

Rule set	Pseudorules	Items in Vertex Table	c
synth1	14 187	29 943	2.11
synth2	21 882	46 185	2.11
rules1	601 512	1 333 060	2.16
rules2	167 120	370 368	2.21
rules3	17 390	42 489	2.44
rules4	22 295	54 474	2.44

Table 6.4: Size of Vertex Table related to the number of pseudorules.

6.6.2 Throughput

The processing time for each packet is split into several steps: The first step is the LPM operation in all dimensions. The time of tree-based LPM algorithms is linear with the size of the dimension. However, the size of each dimension is always known in advance and almost never changes. For example the LPM processing IPv6 address must match at most 128 bits. Therefore, the LPM may be considered as operation with constant time in the scope of packet classification. Moreover, approaches running in truly constant time were already published [16].

The FPGA implementation of spoilers TCAM memory gives one result per clock cycle [9]. The remaining steps of the algorithm are two hash functions f_1, f_2, two vertex table accesses, addition, one rule table access and one match of the original packet with the rule from the rule table. All of these operations run in constant time and therefore, the PHCA runs in constant time.

PHCA was designed for implementation in FPGA or ASIC. The LPM computation can run in parallel in each dimension, because LPMs are independent. The algorithm is designed as a pipeline of simple steps. There are no complex mathematical operations such as division, which are particularly hard to implement in FPGA.

Supposed that all logic can be implemented on-chip to achieve any required throughput (possibly even replicated to achieve it), the overall throughput of the algorithm is determined by the off-chip communication. Only the Vertex Table is too large to fit into on-chip

Rule set	Rule Table	Vertex Table
synth1	74.27	479.08
synth2	135.87	738.96
rules1	33.44	21 289.96
rules2	58.08	5 925.88
rules3	93.98	679.82
rules4	4 350.72	871.58

Table 6.5: Memory size of the Vertex Table and Rule Table [kbit] of PHCA.

FPGA or ASIC memories, and is proposed to be stored in external SRAM. The perfect hash function evaluation requires to read and add two integers from the Vertex Table for each packet. The width of integers must be enough to store the rule number. For example, memory width of 16 bits will support up to 65536 rules. The throughput with commodity FPGA and SRAM is 266 million packets per second (suppose RLDRAM2 running at 533 MHz [4] is used as external SRAM to store the Vertex Table and there is no other obstacle in the algorithm performance). This can be compared to 150 million packets per second throughput of 100 Gbps Ethernet for the shortest 64 B frames.

Chapter 7

Prefix Filtering Classification Algorithm

While the Perfect Hash Crossproduct Algorithm removes the need for storing pseudorules, the size of its data structures still depend on the number of pseudorules. The Prefix Filtering Classification Algorithm (PFCA, [28]), presented in this section, aims to utilize certain common properties of rule sets to lower the number of pseudorules and therefore to reduce the domain of perfect hash function.

7.1 Algorithm Description

The method of lowering the number of pseudorules is based on the observation that many classification rules often contain universal conditions. For example, if the user of a firewall wants to block a specific source IP address, the filtering rule does not specify any destination IP address nor the port number. This means that packet with any destination IP and any port number matches this rule. However, the rule can create many pseudorules because all more specific destination IPs and ports have to be covered.

Figure 7.1 is an example for two fields, where the universal condition in the rule set produces many pseudorules. The situation is even worse for multiple fields, because pseudorules create Cartesian product. Table 7.1 shows that all tested rule sets contain large number of universal conditions. In rules2, most rules contain four universal conditions. There is no clear pattern for universal conditions in rules1. The remaining four rule sets contain rules mostly with one or two universal conditions. Table 7.2 shows numbers of universal conditions in each dimension.

PFCA inserts a *Generalization Stage* (GS) into the classification algorithm after LPM engines (Figure 7.2). If it is applicable, GS is able to replace LPM results (parts of LPM vector) by the ANY value without loosing any information needed for correct packet classification. As a result, number of output combinations is reduced after GS. This will reduce the data structures of the following stages of all crossproduct algorithms. In other words: If GS replaces pseudorule by a rule, then the pseudorule does not need to be treated. Other parts of the algorithm remain the same as in PHCA.

Definition 7.1 (Generalization Rule)**.** *The generalization Rule (GR) is a 3-tuple* $g = (b, v, G)$ *where* b *is an index to the LPM vector,* v *is a value of a particular field of LPM vector, and* G *is a set of indices to the LPM vector.*

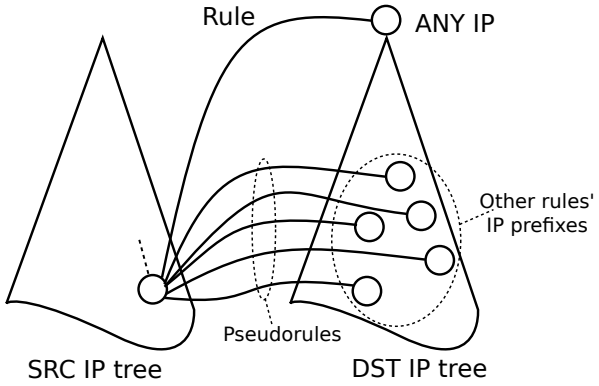


Figure 7.1: One of the most severe causes of pseudorules: ANY values in the rule set.

Rule set	Rules with N universal conditions					
	$N=0$	$N=1$	$N=2$	$N=3$	$N=4$	$N=5$
synth1	0	238	138	22	2	0
synth2	0	440	266	31	3	0
rules1	0	32	19	40	18	1
rules2	0	0	16	6	161	1
rules3	0	165	150	19	1	0
rules4	0	1079	247	0	5	0

Table 7.1: Numbers of universal conditions in rules.

Definition 7.2 (Generalization Rule Effect)**.** *The effect of one GR is: if* $LPM[b] = v$*, then for each index* $i \in G$ *set* $LPM[i] := ANY$*. All GRs may be applied together, their ordering is unimportant.*

This scheme corresponds to the following situation: we know that if a field $LPM[b]$ has a particular value v, then some other fields $LPM[i], i \in G$ are unimportant, because the result of classification is already determined.

It remains to find an algorithm to create GRs. First, all pseudorules must be found by Algorithm 4.1. The algorithm creates a list of pseudorules where all pseudorules corresponding to one rule are stored continually. We call these continual sections *P-blocks*. We can say that the list of pseudorules consists of k P-blocks, where k is the number of rules and the P-blocks are sorted from the highest to the lowest priority. Note that each rule is also added as a pseudorule before all of its pseudorules are added to the list. This leads to the fact that in each P-block, the first pseudorule is the most general of all pseudorules in that P-block, and it is the original rule itself. This ordering is helpful in the next part of the algorithm where GRs are created and some pseudorules are removed.

Algorithm 7.1 identifies situations when the rule defines some field with index d and allows the ANY value in fields with indices from G. Then in certain cases, for all LPM vectors with the same value at index d, values at indices from G may be replaced by ANY value, and the result of the classification is still uniquely determined.

There are several conditions that must be met when creating a GR:

- The rule must contain at least one ANY value. This condition is not explicitly written

Rule set	Universal conditions in dimension				
	SRC IP	DST IP	Protocol	SRC Port	DST Port
synth1	9	90	0	71	374
synth2	9	137	0	121	649
rules1	54	24	32	104	50
rules2	100	91	162	184	162
rules3	2	32	4	275	153
rules4	51	5	5	1244	189

Table 7.2: Numbers of universal conditions in rules.

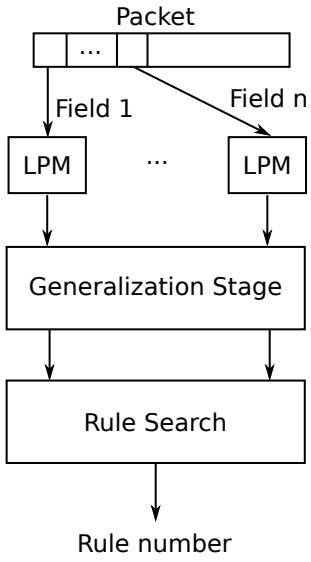


Figure 7.2: Structure of the Prefix Filtering Classification Algorithm.

in the algorithm, because it is implicit: for rules with no ANY value, G would be empty and the GR would make no sense.

- The same value of the field at index d has not appeared earlier in the list of pseudorules. If this condition is not true, we cannot be sure that the value in this field unambiguously determines the correct classification result.
- The ANY value has not appeared at the index d earlier in the list of pseudorules. The reason for this condition is the same as for the preceding one.

7.2 Example

To demonstrate the function of the algorithm, consider rules and pseudorules from Table 7.3 and Figure 7.3.

There are two pseudorules in this example, both of them are specific cases of the rule $R3$. If the value in the Dimension 1 of LPM vector is 101, then the result of classification is already unambiguously determined (because $R3$ has the highest priority). Thus we create Generalization Rule $(1, 101, \{2\})$. The Generalization Stage with this GR will perform

Algorithm 7.1 Creating generalization rules and removing pseudorules.

1: Input: List of pseudorules P.
2: Create empty set of generalization rules GR.
3: **for all** pseudorules $p_{this} \in P$ **do**
4: **for all** dimensions d **do**
5: **if** $p_{this}[d] \neq ANY$ and no previous $p_{prev} \in P$ exists such that ($p_{prev}[d] = p_{this}[d]$ or $p_{prev}[d] = ANY$) **then**
6: Create new generalization rule $g_{new} = (d, p_{this}[d], G)$, where $G = \{i | p_{this}[i] = ANY\}$.
7: Add g_{new} to GR.
8: Remove all pseudorules p_{after} that follow in P after p_{this} where $p_{after}[d] = p_{this}[d]$ and exists $i \in G$ such that $p_{after}[i] \neq ANY$.
9: **end if**
10: **end for**
11: **end for**
12: Output: Set of generalization rules GR, reduced list of pseudorules P.

Rule	Dimension 1	Dimension 2	Priority	Target rule	P-block
R1	*	100	1		1
R2	*	10*	2		2
R3	101	*	3		3
P1	101	10*	3	R3	3
P2	101	100	3	R3	3

Table 7.3: Example rules and pseudorules organized in P-blocks.

substitution of LPM vectors $(101, 10*)$ and $(101, 100)$ by $(101, *)$. Therefore, pseudorules $P1, P2$ are not necessary in the following steps of the classification algorithm and only the original rules need to be stored in this simple example.

7.3 Algorithm Correctness

To show that PFCA correctly classifies the packets, we start with the assumption that the Perfect Hashing Classification Algorithm is correct. Namely we suppose that the Rule Search step from Figure 4.6 is able to obtain correct classification result from LPM vectors with the knowledge of all pseudorules. Therefore we may suppose that the Rule Search step simulates linear search in the list of rules and pseudorules ordered by priority (as defined in Section 3.5).

To show the correctness of the generalization algorithm, we have to show that after processing LPM vectors in the GS (see Figure 7.2), there is still enough information to obtain correct classification result, with the knowledge of the reduced set of pseudorules.

The condition in the line 5 of the Algorithm 7.1 means that if index d of the LPM vector has the value $p_{this}[d]$ and the packet does not match any rule or pseudorule with higher priority, then the classification result is unambiguously known. This is because if the packet should be matched by a higher-priority rule, then it will be matched correctly during the (supposed) linear search in the Rule Search step. Therefore, generalization is

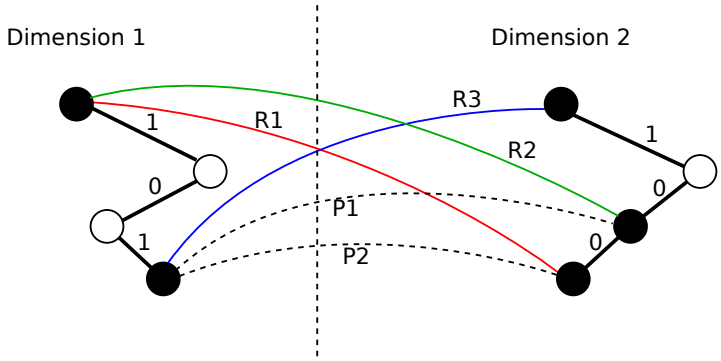


Figure 7.3: Example rules and pseudorules.

performed only if the classification result is unambiguously known.

All removed pseudorules must have the same or lower priority than the actual rule. The condition at the line 8 of the algorithm requires removed pseudorules to be specific cases of the actual rule. And because the correct result of the classification is already known, it is not necessary to store more specific pseudorules of this rule to classify the packet correctly.

7.4 Hardware Implementation

While the implementation of PHCA is rather straightforward, the logic performing the generalization in PFCA deserves more detailed description. The hardware implementation of the Generalization Stage must have enough performance to process packets at the wire speed. The following scheme is proposed:

A set of GRs is split into d subsets, where all GRs with the same index b are in the same subset (GRs are split by dimension). For each field, one *Field Generalization Engine (FGE)* is used. The FGE is a structure similar to Content Associative Memory (CAM): it contains tuples (v, G_v), where v has the same meaning as in the original definition of GR. G_v is the set G, represented as a vector of d bits, which are set to 1 if the bit index is present in G. FGE compares the input word to all values of v and returns the corresponding vector G_v in the case of match. No more than one match in each FGE is possible, because of the construction of GR searching algorithm. The results of all n FGEs are OR-ed to get the resulting replacement bitmap. For every 1 in the replacement bitmap, the LPM result at that index is replaced by ANY value.

The scheme of the Generalization Stage is shown in Figure 7.4.

7.5 PFCA Evaluation

PFCA is a direct improvement of the PHCA algorithm. The throughput of the algorithm is not affected by the optimization, only latency may increase due to added pipeline step.

7.5.1 Memory

PFCA memories differ from PHCA in two ways: There is the Generalization Stage containing Generalization Rules, and the Vertex Table is smaller thanks to it. Other parts (LPM engines, Rule Table, spoilers) remain the same.

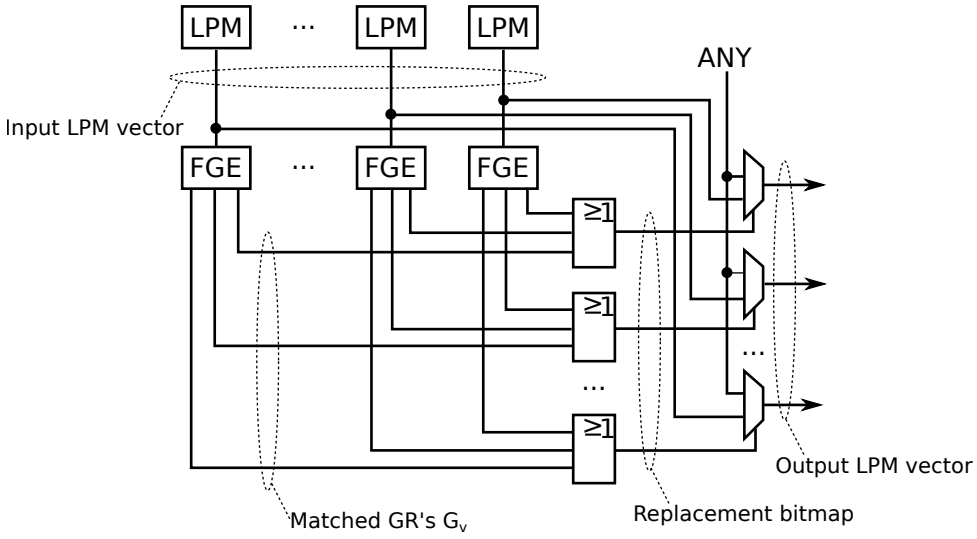


Figure 7.4: Scheme of the Generalization Stage

If the GS is implemented as proposed in Section 7.4, each GR stores only v (prefix value) and G_v (bitmap of dimensions). If we suppose that each LPM engine can store up to 1024 items (the largest prefix set in the benchmark rule sets has 158 items, see Table 5.1), then v is stored at $\log_2(1024) = 10$ bits. The bitmap G_v stores one bit for each dimension, so that one GR will typically use 15 bits of storage.

Table 7.4 shows how many GRs can be created in each dimension. It can be seen that for rules2, only one GR was found. Table 7.5 analyzes how the number of pseudorules was reduced in PFCA compared to plain PHCA. Rule set rules2 shows only 1.2 % improvement. Table 7.6 shows the memory size of GS and Vertex Table for all tested rule sets.

Rule set	Generalization Rules in dimension					
	SRC IP	DST IP	Protocol	SRC Port	DST Port	Sum
synth1	49	29	0	2	0	80
synth2	54	33	0	4	0	91
rules1	0	0	2	0	10	12
rules2	0	1	0	0	0	1
rules3	43	0	1	0	0	44
rules4	7	77	1	0	0	85

Table 7.4: GRs found in rule sets.

The main disadvantage of PFCA is its great dependence on the rule set: While the reduction of pseudorules can be as good as 94.9 % for the rule set rules3, it can also be only 1.2 % for the rule set rules2. This is caused by the arrangement of universal conditions in the rule set.

Specifically for the rule set rules2, the issue is that the first rule in the rule set (the rule with the highest priority) defines only the destination IP address, leaving universal conditions in all other dimensions. One GR is created from this rule, but also all other dimensions than destination IP address are banned for standing as b in the GR (see Definition 7.1) by the condition at the line 5 of Algorithm 7.1. The second rule in rules2 defines

Rule set	Pseudorules before	Pseudorules after	Ratio
synth1	14 187	12 493	0.880
synth2	21 882	19 199	0.877
rules1	601 512	118 278	0.196
rules2	167 120	165 129	0.988
rules3	17 390	895	0.051
rules4	22 295	5 225	0.234

Table 7.5: Reduction of pseudorules in PFCA.

Rule set	GS [bit]	Vertex Table [kbit]
synth1	1 200	364.60
synth2	1 365	680.76
rules1	180	440.63
rules2	15	5 310.91
rules3	660	26.09
rules4	1 275	168.03

Table 7.6: Memory size of the GS and Vertex Table of PFCA.

only the source IP address, leaving universal conditions in all other dimensions. Not only the GR can not be created from this rule, but also the destination IP address dimension is banned thanks to this rule. Therefore, no other GRs can be created after processing only the first two rules of the rule set.

The following algorithm aims to achieve better stability.

Chapter 8

Prefix Coloring Classification Algorithm

The Prefix Coloring Classification Algorithm (PCCA, [40]), described in this chapter, aims to overcome the disadvantage of PFCA. The pseudorules reduction used in PCCA gives more stable results across different rule sets.

8.1 Introduction

Let us focus on the fact that the LPM operation is performed independently for each field in decomposition-based packet classification algorithms such as PHCA and PFCA. (see Figure 4.6). The advantage of this scheme is the strong potential for parallel computation. On the other hand, LPM results are logically related – only certain combinations of LPM results form a rule, the rest of them are unwished pseudorules. Thus, the knowledge of LPM result from one dimension *should* affect LPM result in other dimensions. Figure 8.1 shows an example situation when the knowledge of LPM result from one dimension has impact on the LPM of the other dimension: When the Dimension 2 LPM result is $00*$, then there is no need to continue searching below prefix $1*$ in the Dimension 1 LPM. The result 101 would not bring any new information significant for the packet classification, because no other (pseudo) rule is reachable. The aim of this chapter is to find an effective way of exchanging information among dimensions before the LPM results reach the Rule Search stage.

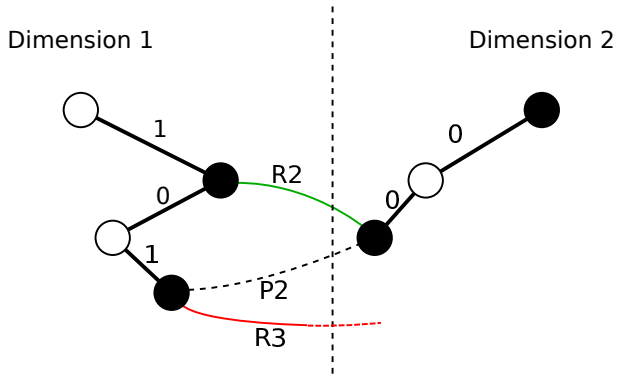


Figure 8.1: Motivation for communication between LPMs. There is no need to traverse the Dimension 1 trie below $1*$ if the Dimension 2 LPM result is $00*$.

The example of algorithm which takes one LPM result into account when performing the other LPM is the Grid-of-Tries [46]. In the basic version of that algorithm, there is one second-level trie for each valid result of the first-level trie. This scheme (generalized in Figure 8.2) does not scale well. The implied sequential processing of dimensions is not an issue, because it is easily pipelined. The worse fact is that the number of tries at lower levels can grow exponentially. There is one second-level trie for each result of the first level trie, and there is one third-level trie for each result of each of the second-level tries etc. Also, Grid-of-Tries selects some particular ordering of dimensions, making the communication between LPMs only unidirectional. The example shown in Figure 8.1 would not work in Grid-of-Tries in Figure 8.2, because it requires the information from Dimension 2 to be used in Dimension 1.

However, the idea of one LPM result affecting (and being affected by) other dimensions' LPM results is the key to the PCCA presented in this chapter.

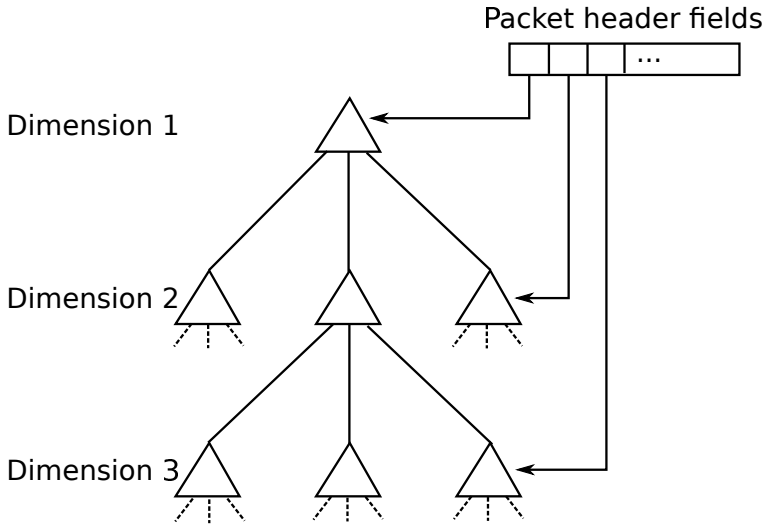


Figure 8.2: Generalized scheme of the Grid-of-Tries algorithm.

8.2 Algorithm Description

In the Grid-of-Tries algorithm the tries at lower levels always store some subset of the full prefix set of the corresponding dimension. PCCA delays the communication among dimensions *after* the LPM operation, so the LPM engines must return maximum amount of data. Let's suppose that the LPM operation is modified to return *all* matching prefixes, not only the longest one. The newly added Color Processing Stage aims to select from these prefixes only combinations which are in the rule table. This selection would remove all pseudorules, making the Rule Search step easier.

Let each prefix p_1 contain a precomputed bitmap for all other dimensions. There is one bit for each prefix of each dimension in the bitmap. The bit stored in prefix p_1 corresponding to prefix p_2 is set to 1 if prefixes p_1 and p_2 appear together in some rule. Otherwise the bit is set to 0. We call this bitmap the *full bitmap*. Each LPM result (prefix) now contains an information about "allowed" and "suppressed" prefixes from other dimensions. From all LPM results, only allowed prefixes are then used to create the LPM vector which is then passed to the following stages of the algorithm.

It is possible to remove almost all pseudorules this way, because most unwanted LPM vectors are filtered away. Not *all* pseudorules are removed, because the information about rules priority is missing in prefixes. (see Figure 8.3).

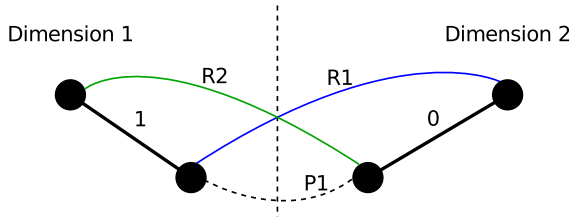


Figure 8.3: Pseudorule $P1$ can not be removed without the information about rules priority.

The full bitmap has a disadvantage in adding large memory overhead to the LPM results. Number of prefixes may be large, therefore LPM results table would have to use very wide data words. Table 5.1 shows that numbers of prefixes are too large for full bitmaps in larger rule sets. Moreover, sizes of full bitmaps vary with each rule set.

Instead of using full bitmaps, fixed amount of *groups of prefixes* are created in PCCA to limit the size of bitmaps. Only small bitmaps for groups are stored. Grouping of prefixes may be based on different criteria. There is always some *implicit* information in the prefixes: Each prefix has its length, which can be used directly as a group number. The length may also be divided by a constant to create more coarse-grained grouping and thus smaller bitmaps. Number of parents in the LPM tree is called prefix *nesting* and can also be used for prefix grouping.

Explicit grouping information can be added to the prefixes: we assign an abstract *color* property to each prefix. Number of colors is set to be much smaller than the number of prefixes. Instead of carrying large full bitmaps of prefixes, each prefix contains its own color and only a small bitmap of allowed colors for each other dimension.

Explicit prefix coloring is the most general option, because all other grouping criteria can be simulated by proper assignment of colors. Therefore, only explicit grouping by colors is used in the following text.

The bitmap stored in prefixes is called the *Allowed Colors Bitmap* (ACB). Instead of returning all matching prefixes, the LPM operation returns the longest matching prefix for each color. Also, it returns the *Aggregate Allowed Colors Bitmap* (AACB) which is a bitwise logical disjunction (OR) of all ACBs observed during the LPM tree descent.

The modified processing pipeline of the classification algorithm is in Figure 8.4. The Color Processing Stage works like a filter, which checks each input prefix against AACBs from other dimensions. Only one prefix for each dimension may pass through it. The Color Processing Stage selects the longest prefix from prefixes that are allowed by all AACBs. Operations of the Color Processing Stage are shown in detail in Algorithm 8.1.

One simple method of assigning colors to prefixes is presented here, further discussion is given in Section 8.5. This method is designed to achieve the balanced distribution of prefixes among colors: Prefixes are sorted by length and then assigned colors sequentially with repetition (i.e. 0, 1, 2, 3, 0, 1, ...).

The method of filling ACBs in prefixes is straightforward: At the beginning, all bitmaps contain zeros. Then for each rule r and each prefix $r.d$ of rule r, set bits in the ACB to one, such that bitmaps allow colors of all other prefixes of the rule r.

It remains to find an algorithm that generates pseudorules. Due to colors and color bitmaps, majority of pseudorules are suppressed. However, some pseudorules are gener-

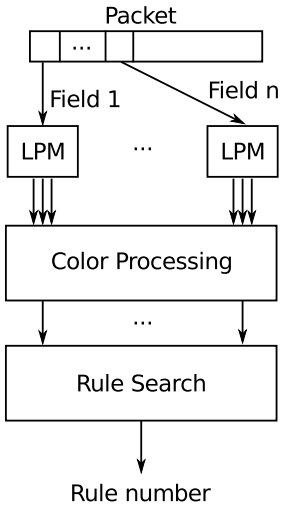


Figure 8.4: Improved scheme of decomposition algorithm.

ated for most rule sets. The experience with previous algorithms is that generating all pseudorules during the software precomputation phase may be highly time-consuming operation. Therefore it is undesirable to generate all pseudorules and then remove some of them. Instead, an algorithm that directly generates only pseudorules which have to be considered in the Rule Search Stage is presented (Algorithm 8.2).

Algorithm 8.1 Operations of the Color Processing Stage.

Input: AACB for each dimension, prefix for each dimension and color.

for all dimensions d **do**
create the *Present Colors Bitmap* (PCB_d) where bits correspond to colors in dimension d. Each bit of PCB_d is set to 1 if some prefix with that color was returned by the LPM, and to 0 otherwise.
end for
for all dimensions d **do**
Final Allowed Colors Bitmap $\mathrm{FACB}_d \leftarrow$ bitwise_and(PCB_d, corresponding AACBs from all other dimensions).
end for
if some FACB contained all zeros **then**
Packet matches no rule.
else
Create empty output LPM vector V.
for all dimensions d **do**
Add the longest matching prefix allowed by FACB_d to V.
end for
end if

Output: LPM vector V

Algorithm 8.2 Pseudorules generating with respect to colors.

Input: Rules with prefixes containing ACBs and their own color.

Create empty list of pseudorules P.
The rule set is traversed from the highest to the lowest priority:
for all rules r **do**
for all dimensions d **do**
$L_d \leftarrow$ list of all prefixes from dimension d matching rule r. In this list, there is prefix $r.d$ from rule r, and all more specific (longer) prefixes from other rules.
end for
A decision tree is traversed. Each tree level corresponds to one dimension d, dimension ordering is unimportant. Tree edges are prefixes from L_d. The tree is traversed by a depth-first traversal algorithm. Descent is performed only if colors and ACBs in prefixes allow the combination of prefixes from the root of the tree to the current leaf.

if the lowest tree level is reached **then**
if (the combination of prefixes from root to leaf) $\notin P$ **then**
Add the prefix combination into P. (r is also added by this operation)
end if
end if
end for

Output: List of pseudorules P

8.3 Color Processing Example

Example rule set from Figure 4.10 is used to demonstrate function of the algorithm. Figure 8.5 shows how colors are assigned to prefixes, and how the color bitmaps are filled. Bitmaps are shown as sets. Two colors are used in each dimension. Figure 8.6 is a decision tree, according which the pseudorules are generated. In our example, one pseudorule is generated. Tree paths $(f1p2, f2p1)$ and $(f1p2, f2p2)$ are not examined, because prefix $f1p2$ has color $c1$, and color bitmaps of prefixes $f2p1$ and $f2p2$ do not allow color $c1$.

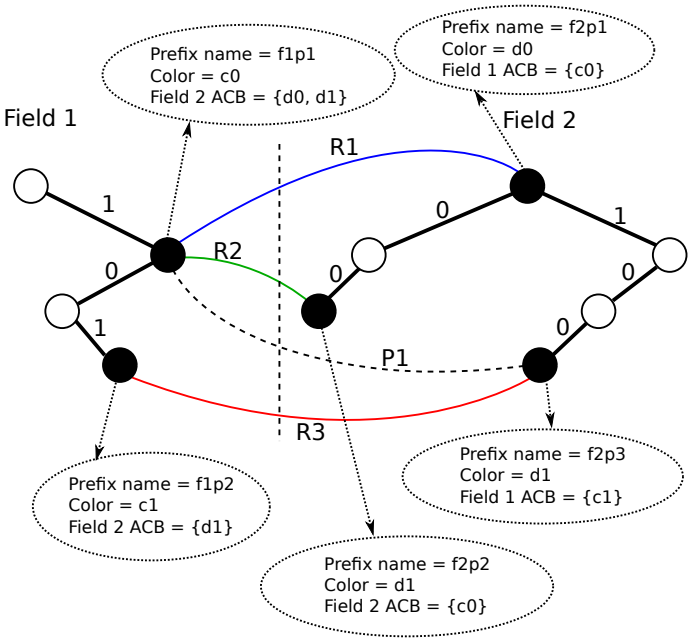


Figure 8.5: Prefix colors and color bitmaps. Prefixes are assigned names for easier presentation.

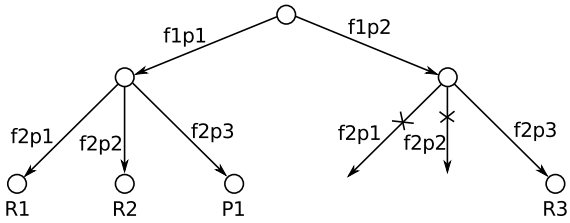


Figure 8.6: Decision tree for generating pseudorules.

We continue by showing how a packet is processed. Suppose packet with header fields $(101, 000)$. In the Field 1, the LPM returns:

- For each color, name and length of the longest matching prefix:

Color	Prefix	Length
$c0$	$f1p1$	0
$c1$	$f1p2$	3

- AACB gathered from visited prefixes: $\{d0, d1\}$

In the Field 2, the LPM returns:

- For each color, name and length of the longest matching prefix:

Color	Prefix	Length
$d0$	$f2p1$	0
$d1$	$f2p2$	2

- AACB gathered from visited prefixes: $\{c0\}$

The result in the Field 1 is $f1p1$, because only color $c0$ is allowed by Field 2 results, and $f1p1$ is the longest matching prefix of this color. The result in the Field 2 is $f2p2$, because both colors $d0$ and $d1$ are allowed by Field 1 results, and $f2p2$ is the longest matching prefix.

To sum up the example, the prefix colors and color bitmaps avoid the unwished combination of LPM results $(f1p2, f2p2)$, which would be returned in case of unmodified LPM operation. Instead, the output of the Color Processing Stage is the LPM vector $(f1p1, f2p2)$, which corresponds to the rule $R2$.

8.4 PCCA Evaluation

8.4.1 Throughput

Similar to PFCA, PCCA is also improvement of the basic PHCA algorithm. Analysis of the algorithm throughput is based on the analysis of PHCA. The modification of the LPM algorithm required by PCCA is to return more prefixes instead of only the longest matching one. This can be implemented by having the data word wide enough to fit all prefixes. Each prefix must also store its ACB. However, these modifications should not affect the LPM throughput.

The Color Processing Stage uses only simple logic operations (Algorithm 8.1). Implementation of the Color Processing Stage in Virtex-6 FPGA logic consumes 1364 LUT-FlipFlop pairs, and can run at 262 MHz (after synthesis for 5 dimensions and 8 colors). It only adds four cycles of latency. Moreover, this small logic can be easily replicated to achieve the required throughput.

8.4.2 Memory

PCCA is a modification of PHCA at several points:

- LPM engines return one result for each color, and each prefix also stores its color and ACB.
- The Color Processing Stage selects one LPM result in each dimension. This is only combinational logic with no memory.
- The Vertex Table is smaller.

The memory added to prefixes depends on number of colors. For example, if we use 8 colors, then each prefix must store its color (3 bits) and 8-bit bitmap for each of the remaining dimensions. That is 35 bits per prefix. Table 8.1 shows the memory added to prefixes in LPM for the different number of colors.

Rule set	Colors				
	4	8	16	32	64
synth1	2 142	4 165	8 092	15 827	31 178
synth2	2 466	4 795	9 316	18 221	35 894
rules1	2 232	4 340	8 432	16 492	32 488
rules2	3 276	6 370	12 376	24 206	47 684
rules3	2 268	4 410	8 568	16 758	33 012
rules4	5 202	10 115	19 652	38 427	75 718

Table 8.1: Memory added to LPM Stage for different numbers of colors (bits).

Table 8.2 shows the size of the Vertex Table. It can be seen that adding more colors helps to avoid pseudorules and thus to reduce the size of the Vertex Table. One exception is rule set rules2, which does not improve when more than 16 rules are used.

Rule set	Colors				
	4	8	16	32	64
synth1	383.58	301.84	133.48	86.62	55.23
synth2	656.30	537.72	295.24	185.84	109.31
rules1	12 063.78	3 741.00	3 186.38	392.38	144.97
rules2	21 974.40	1 321.23	549.68	549.68	549.68
rules3	455.15	180.33	75.87	39.80	13.93
rules4	625.44	527.71	319.00	170.70	111.58

Table 8.2: Size of the Vertex Table for different numbers of colors (kbits).

Graph in Figure 8.7 shows the sum of LPM and Vertex Table memory sizes. To compare the memory to the original PHCA, the graph contains also values for 0 colors, which are the sizes of the Vertex Table from Table 6.5 (plain PHCA).

8.5 Prefix Coloring Strategies

The optimal coloring for the given number of colors and the given rule set is the coloring which results in the smallest number of pseudorules in Algorithm 8.2. It is unknown to the author whether the optimal coloring can be found by an algorithm better than the exhaustive search. The exhaustive search evaluates all possible colorings and finds the one which results in the lowest number of pseudorules. It is clear that such algorithm is not useful in practice. Consider for example the smallest rule set rules1 used in this work, colored by 8 colors. The number of possible colorings is

$$8^{28+48+4+6+40} \approx 6.1 \times 10^{113}$$

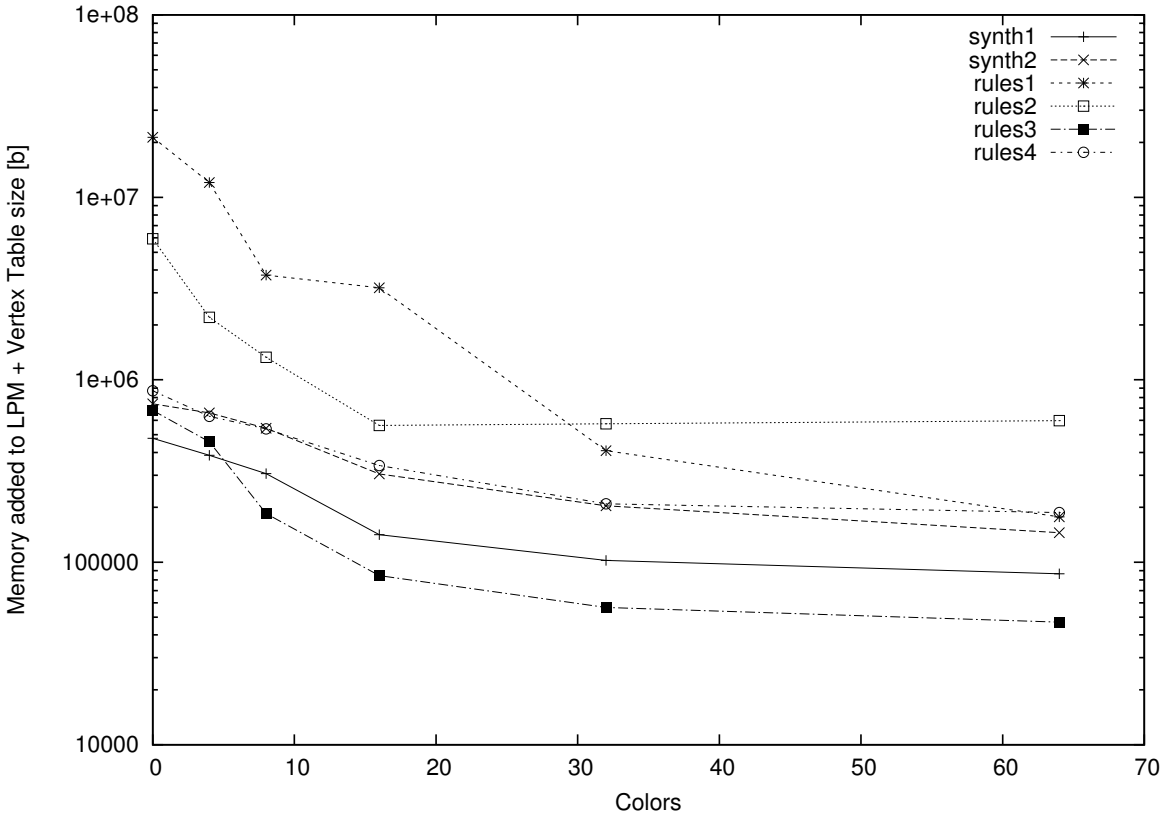


Figure 8.7: Memory of the PCCA algorithm.

Even if some impractical colorings are skipped, it is clear that the exhaustive search can not be used in practice. For the example of impractical coloring consider the Protocol field of rules1, where only 4 prefixes are found. If there are more colors than prefixes, only one coloring (the one that assigns different colors to all prefixes) makes sense.

This section attempts to evaluate several possible coloring strategies in order to find some good (yet not optimal) solution to the coloring problem. The simple method of assigning colors to prefixes described in Section 8.2 is only one of many possible approaches. It is in fact very simple heuristic designed to achieve balanced use of all colors. The next prefix coloring heuristic discussed in this work is the random assignment. As was mentioned, prefix grouping may be also derived from some implicit information contained in the LPM tree, such as the prefix length or the nesting level. In such cases, prefixes do not need to store their own color, only the bitmaps.

Table 8.3 shows number of pseudorules for various coloring strategies. As was explained in Section 6, the number of pseudorules has direct impact to the size of Vertex Table. The Table 8.3 contains results for the following strategies:

- *Default*: 8 colors assigned sequentially to prefixes sorted by length, with overflow.
- *Best Rnd*: The best result from 10 runs of random assignment of 8 colors.
- *Avg Rnd*: The average value from 10 runs of random assignment of 8 colors.
- *Len*: The prefix color equals the prefix length. For example 33 colors (0-32) are used for IPv4 addresses.

- *Len/2*: The prefix color equals the prefix length divided by two.
- *Len/4*: The prefix color equals the prefix length divided by four.
- *Nesting*: The prefix color equals the prefix nesting level (number of parent prefixes in LPM tree).
- *Full*: Each prefix has unique color in its dimension.

Rule set	Strategy							
	Default	Best Rnd	Avg Rnd	Len	Len/2	Len/4	Nesting	Full
synth1	7 721	5 630	7 617	3 959	5 994	8 298	13 265	1 558
synth2	13 755	13 025	14 205	7 139	10 052	13 409	18 780	3 083
rules1	116 317	93 606	152 669	184 234	191 298	203 031	209 315	4 508
rules2	35 487	28 607	45 168	14 764	28 607	56 293	83 564	14 442
rules3	5 086	4 571	6 244	17 380	17 382	17 382	17 390	413
rules4	12 856	10 438	11 966	17 431	17 443	17 436	17 196	1 586

Table 8.3: Number of pseudorules for different color assignment strategies.

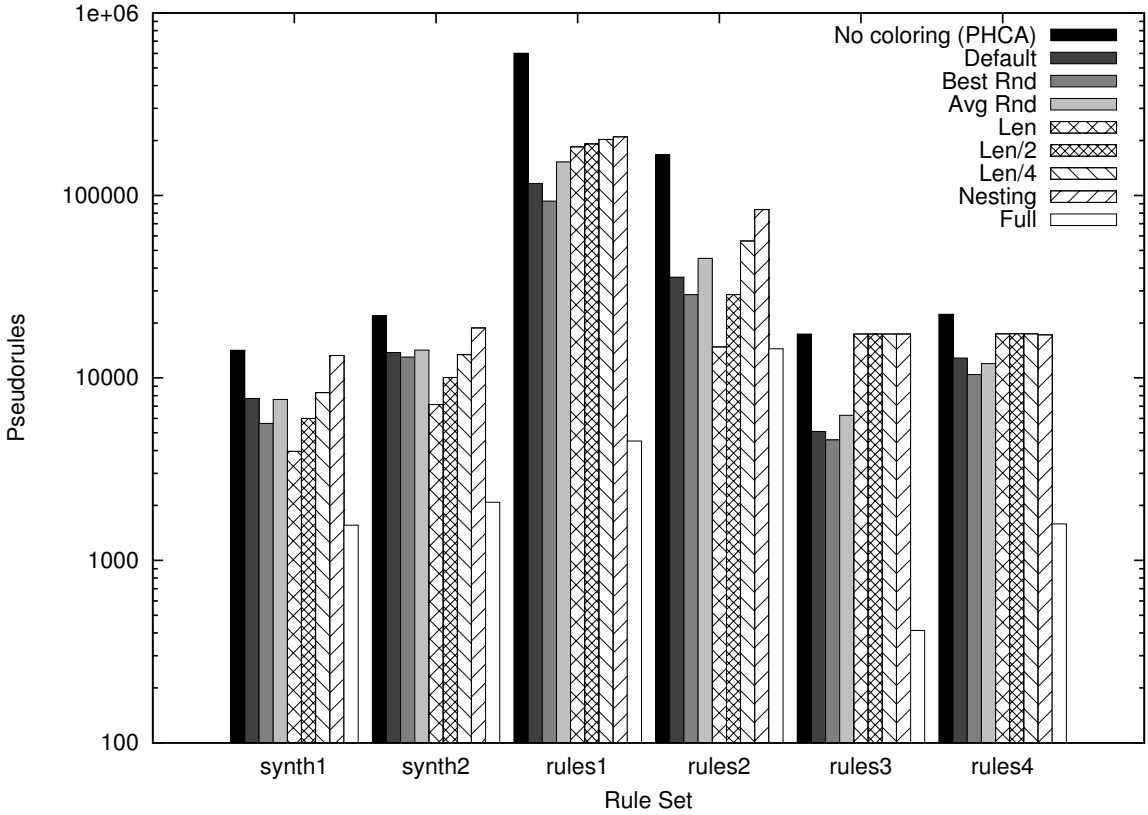


Figure 8.8: Different coloring strategies of PCCA.

Figure 8.8 shows the results from Table 8.3 in graphical form. Number of pseudorules without coloring (plain PHCA) is also displayed. Note the logarithmic vertical scale.

Several conclusions can be deduced from the Table 8.3. The most important finding is that the Default coloring strategy is not significantly better than random coloring. The best random color assignment from 10 runs is better than the default strategy for all rule sets.

Graph 8.9 is a histogram for number of pseudorules obtained by 100 runs of random coloring with 8 colors in the rule set rules3. Results are grouped into intervals of width 200. The results can not be better than the full coloring (413 pseudorules). The results also can not be worse than PHCA (no coloring, 17 390 pseudorules).

The dotted line suggests normal distribution with $\mu = 7054$ (average) and $\sigma = 1568$ (standard deviation). While the normal distribution may not perfectly model the discussed phenomenon, it can give us some hint about the probability of getting significantly better results than those obtained in our 100 runs.

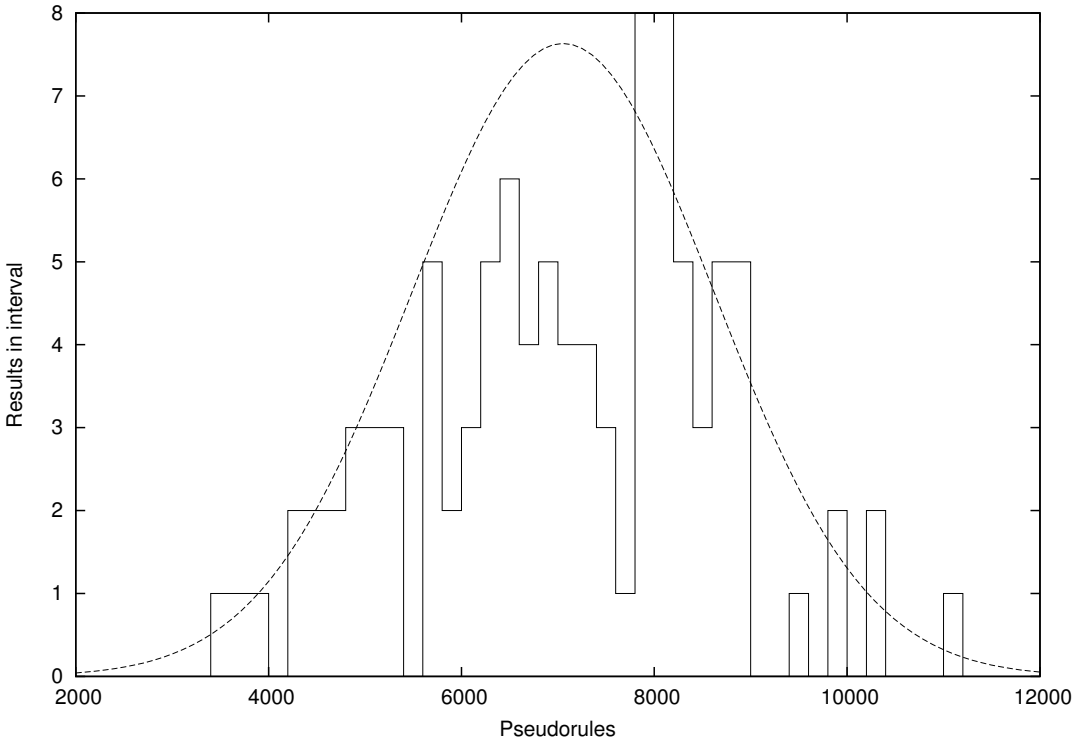


Figure 8.9: Histogram of numbers of pseudorules in 100 random runs.

From the strategies compared here, full coloring certainly generates the lowest number of pseudorules. This is however compensated by very large bitmaps. Using the prefix length (or more coarse-grained length/2) as prefix color gives better results for some rule sets and does not require to store the prefix color, but we have to bear in mind that this approach requires 33 (length) or 17 (length/2) colors for IPv4 addresses. The nesting level is limited in most rule sets, which results in using only few colors and generally more pseudorules. The random coloring gives similar results to the default strategy in average, and achieves better results if the best of several runs is selected. However, more attempts to find good coloring may become time-consuming.

Chapter 9

Multi Subset Prefix Coloring Classification Algorithm

All three previous chapters introduce new strategies to lower memory requirements of decomposition methods. PHCA uses perfect hashing to avoid storing pseudorules, however the number of pseudorules still directly affects the Vertex Table size. PFCA finds and applies Generalization Rules to lower the number of pseudorules. PCCA filters prefixes by colors to lower the number of pseudorules. All three algorithms remove spoilers to further reduce memory. Therefore, PHCA uses two and PFCA and PCCA three optimization strategies. At this point, one may ask whether other combinations of existing optimization techniques are possible.

9.1 Introduction

The Multi Subset Prefix Coloring Algorithm (MSPCCA) aims to combine PCCA with MSCA [17]. MSCA is the first algorithm to introduce the concept of pseudorules. It also proposes the technique of spoilers removal. But the most important optimization of MSCA is the division of rule set into subsets. The algorithm exploits the fact that the sum of Cartesian products of small sets is much smaller than the single Cartesian product of large sets.

MSPCCA employs four optimization techniques:

1. Spoilers removal in a separate algorithm branch.
2. Division of rule set into subsets.
3. Prefix coloring and subsequent filtering.
4. Perfect hash function construction.

To combine all four techniques into a single algorithm, their ordering must be specified. While PCCA uses the techniques in the order 1, 3, 4, MSPCCA inserts the division of rule sets as the second step.

Because MSCA contains several rule sets, matching single packet in all of them would slow down the processing. MSCA therefore uses additional set membership query implemented by the Bloom filters [8] to filter out unnecessary rule table accesses. This filtering is also included in MSPCCA. It is advantageous to use Bloom filters *after* the color filtering, because less items must be stored in Bloom filters in that phase of the algorithm.

The classification engine has the final structure shown in Figure 9.1. (Spoilers branch is not shown.) After the LPM is computed over all fields, Color Processing blocks filter out impossible prefix combinations. A Bloom filter for each subset is then queried for the presence of the prefix combination in the respective subset. Only in the case of positive result the Perfect Hash Function is computed to obtain the pointer to the rule table. The packet is then matched to the selected rule. In case of match, the selected rule is the output, otherwise the default rule is applied. In parallel to the main algorithm there is a separate branch for classification of spoilers. The end of the algorithm performs simple priority resolution between the two branches.

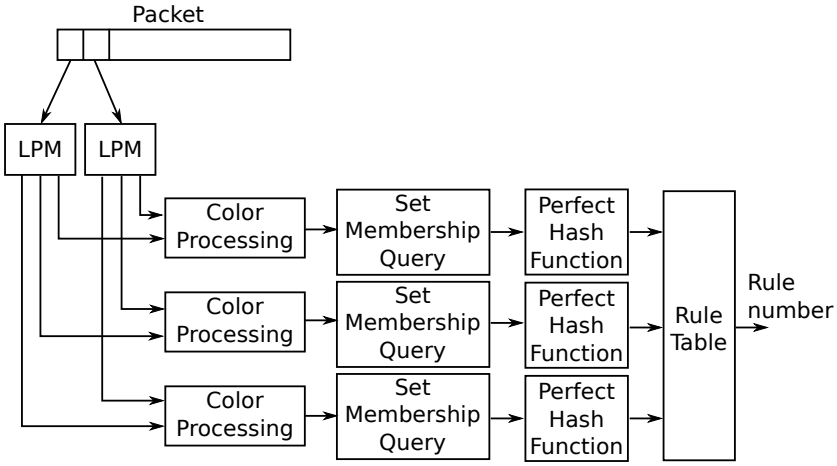


Figure 9.1: Structure of algorithm combination.

In parallel pipelined hardware implementation, several further optimizations are possible, compared to Figure 9.1. Supposing that MSCA is able to avoid match of single packet in multiple subsets, only one instance of the Perfect Hash Function logic is needed. The Vertex Table used by Perfect Hash Function is separate for each subset. The Vertex Table is also the only part of the algorithm that is stored in the external memory. The Perfect Hash Function reads two 16-bit integers from the external memory for each packet.

Similar to previous algorithms, MSPCCA works in two phases – precomputation and the classification itself. The precomputation phase is composed of separate methods as follows: After removing spoilers, MSCA splitting algorithm is used to split the rule set into subsets. Then for each subset, PCCA coloring algorithm is used to assign colors to prefixes and to create the reduced set of pseudorules. The Bloom filters are then filled with the rules and pseudorules of each subset. The last precomputation step is building the perfect hash function for each subset.

The contribution of MSPCCA is the fact that all four optimization techniques are successfully integrated into single algorithm. Spoilers removal, division into subsets and Color Processing contribute to lower the number of pseudorules, while the Perfect Hash Function avoids storing the pseudorules. Therefore it is expected that the new algorithm will require less memory than its predecessors.

9.2 MSPCCA Evaluation

9.2.1 Throughput

MSPCCA is different to PCCA in the fact that multiple instances of the Color Processing Stage are present. Since these instances are independent, they can run in parallel. The other difference is the set membership query, implemented by Bloom filters. Bloom filter requires the computation of several hash functions and access to several bit locations in the on-chip memory. Suitability of implementation of Bloom filters in hardware was shown in many works before [17, 16, 34].

The external memory bandwidth is defined by the *faster* of the two original algorithms – PCCA. Supposing that MSCA is able to avoid match of single packet in multiple subsets, only two 16-bit accesses to the external memory are required to classify a packet in MSPCCA. The algorithm speed is not limited by slower MSCA.

9.2.2 Memory

Table 9.1 shows numbers of rules and pseudorules generated by MSCA. Eight spoilers were removed, three subsets were used, each with eight colors. It is worth noting that the numbers of rules and pseudorules in MSPCCA are getting close to the lower bound, which is the number of rules after expansion of ranges to prefixes (Table 5.1). Rule set rules2 even generates no pseudorules: it contains 184 rules after range expansion, one of them is universal, eight others are removed as spoilers and 175 remain.

Rule set	MSCA	PCCA	MSPCCA
synth1	4 223	7 721	569
synth2	7 623	13 755	1188
rules1	6 003	116 317	610
rules2	415	35 487	175
rules3	547	5 086	295
rules4	2 323	12 856	1645

Table 9.1: Numbers of rules and pseudorules in MSPCCA.

MSPCCA stores several data structures:

- Rule Table stores all rules.
- Vertex Tables store the perfect hash function data for each rule subset.
- Bloom filters are used for the Set Membership Query Stage for each rule subset.
- Color and color bitmaps are stored for each prefix, as well as its membership in rule subsets. (Each subset has different prefix set.)

Table 9.2 shows the sizes of all memory components.

Rule set	Added to prefixes	Bloom filters	Vertex Tables	Rule Table
synth1	12.39	18.00	18.24	74.27
synth2	14.57	37.60	44.51	135.87
rules1	8.52	19.31	21.68	33.44
rules2	12.50	5.54	5.32	58.08
rules3	10.16	9.35	8.91	93.98
rules4	22.72	52.06	58.50	435.07

Table 9.2: Size of different memories of the MSPCCA (kbits).

Chapter 10

Results

This chapter provides overall results of all four presented algorithms. The algorithms are compared to the MSCA, which takes similar approach and therefore can be used for a fair comparison. MSCA is also a candidate for 100 Gb/s solution.

For the illustration of the LPM stage memory, Table 10.1 shows the size of memory needed for Tree Bitmap implementation of LPM for source and destination IP and port conditions. 16 bit pointer width is considered, because the largest prefix set in the test (rules4 source IP) generated up to 337 Tree Bitmap nodes, so that 16 bit pointer should suffice even for much larger prefix sets. Protocol field is not evaluated, because the dimension size is only 8 for protocol, and therefore it can be easily handled by a directly addressed table with 256 items. Dashes (-) in the table represent prefix sets with one prefix only – no LPM is needed in that case. Table 10.1 is common for all five algorithms evaluated in this section (MSCA, PHCA, PFCA, PCCA, MSPCCA).

	Source IP			Destination IP		
Rule set	$s = 3$	$s = 4$	$s = 5$	$s = 3$	$s = 4$	$s = 5$
synth1	1 147	1 316	1 975	1 364	1 410	1 896
synth2	1 271	1 457	1 975	1 147	1 363	1 817
rules1	2 325	3 243	3 792	2 387	3 948	4 108
rules2	5 797	8 507	10 112	5 828	8 554	10 062
rules3	1 426	2 961	2 765	4 805	7 332	8 058
rules4	9 920	15 839	17 459	2 046	5 029	3 950
	SRC Port			DST Port		
synth1	496	705	869	-	-	-
synth2	496	705	869	-	-	-
rules1	527	705	869	2418	3525	4108
rules2	-	-	-	341	423	474
rules3	-	-	-	1488	1974	2607
rules4	-	-	-	3627	4700	6004

Table 10.1: Memory size of the Tree Bitmap implementation for different strides s (bits).

Table 10.1 shows that increasing stride increases the amount of memory in most cases (with only two exceptions: Source IP for rules3 and Destination IP for rules4). On the other hand, higher stride means smaller tree depth and thus less memory accesses and

faster operation.

Table 10.2 and Figure 10.1 show the overall amount of memory for different algorithms and rule sets. The memory for LPM operation is not included, because all algorithms share this step and memory for LPM is measured in the previous table. However, MSCA, PCCA and MSPCCA algorithms add some extra memory to LPM, compared to the other algorithms. This additional memory is included in table 10.2. Eight spoilers are removed in each algorithm. For MSCA and MSPCCA three subsets are used and the probability of Bloom filters' false positive is set to 0.005. Eight prefix colors in each dimension are used for the PCCA and MSPCCA.

Rule set	MSCA	PHCA	PFCA	PCCA	MSPCCA
synth1	1 622.91	553.36	365.80	227.51	122.91
synth2	2 928.36	874.83	682.13	601.23	232.91
rules1	2 303.81	21 362.40	4 403.81	3 347.68	82.96
rules2	162.26	5 983.96	5 310.92	1 075.99	81.45
rules3	211.68	865.93	26.75	260.02	122.41
rules4	898.77	1 306.65	169.30	814.27	568.66

Table 10.2: Memory size of the Rule Search Stage (kbits) for algorithms.

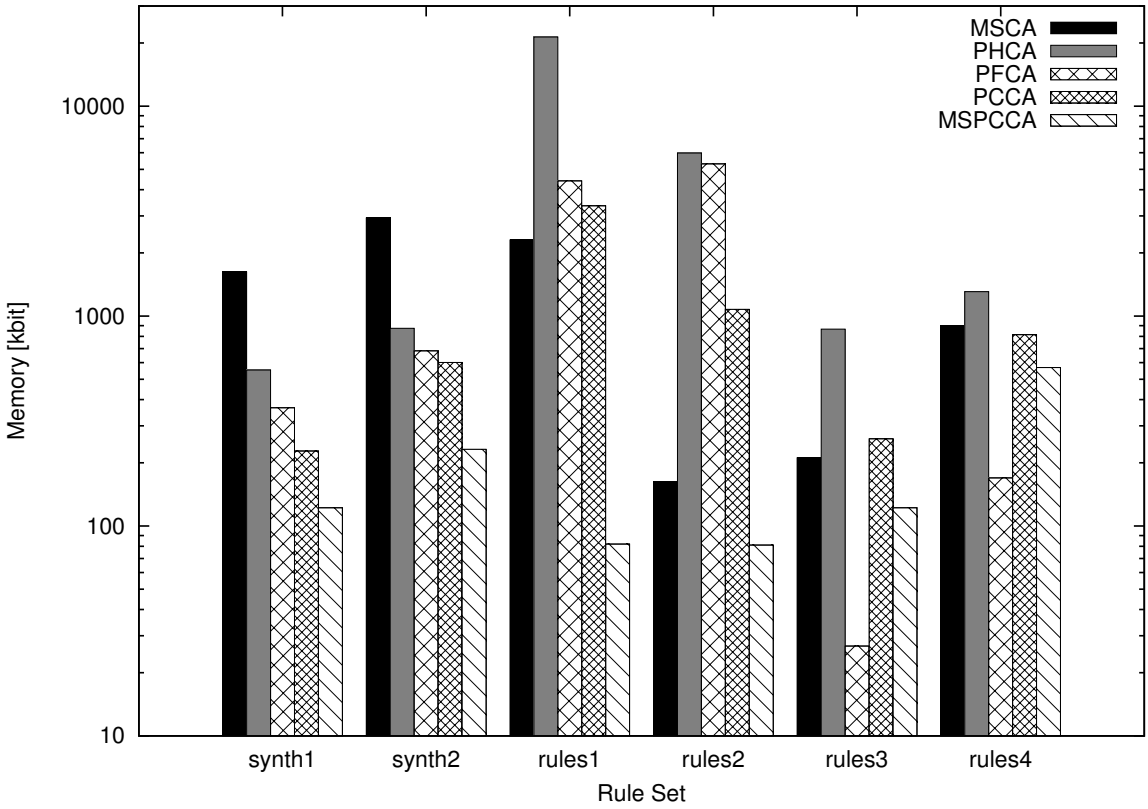


Figure 10.1: Overall memory results for selected algorithms.

MSCA performs well on real-life rule sets rules2 and rules3, but has the worst results

from all compared algorithms for synthetic rule sets. PHCA has poor results for rules1 and rules2 and these rule sets are hard also for PFCA and PCCA. PFCA perform very well on rules3 and rules4, while all other rule sets consume the least memory in the MSPCCA. Rule set rules1 is hard for both MSCA and PCCA, while the combination of both – MSPCCA handles this rule set very well.

Table 10.2 however does not reflect the throughput of the algorithms. While the MSCA must fetch whole rule from the external memory to classify a packet, other algorithms use external memory only to compute the perfect hash function and are therefore better suited for high speed networks. The original paper describing MSCA [17] claims the throughput of 38 million packets per second with the rule table stored in one 300 MHz SRAM. If we consider the same RLDRAM running at 533 MHz as used in other calculations in this thesis, we can suppose that MSCA achieves the throughput of 67 million packets per second. This is well under 266 million packets per second for PHCA, PFCA, PCCA and MSPCCA.

We introduce the *memory to speed ratio* that takes the speed into account. The size of required memory is divided by the number of rules and the algorithm throughput to get the *bits per rule and million packets per second*. Table 10.3 compares the algorithms using this metric. PCCA has the lowest average memory to speed index and also its worst result (rules1) is the lowest of all measured algorithms.

Rule set	MSCA	PHCA	PFCA	PCCA	MSPCCA
synth1	110.60	9.49	6.27	4.76	2.11
synth2	110.93	8.34	6.50	5.73	2.21
rules1	333.83	779.70	160.73	122.18	3.02
rules2	13.99	130.03	115.40	23.38	1.77
rules3	11.48	10.57	0.36	3.55	1.67
rules4	10.78	3.94	0.51	2.46	1.71
average	98.60	157.01	48.30	27.01	2.08

Table 10.3: Memory to speed index.

PHCA is the first algorithm that achieves the 100 Gb/s throughput, but at the cost of inefficient memory utilization, compared to older MSCA. PFCA maintains the throughput of PHCA and reduces the memory significantly. PCCA reduces memory consumption with better results than PFCA in most cases. MSPCCA combines MSCA and PCCA to achieve high throughput and to improve the memory to speed ratio by the order of magnitude in average.

Chapter 11

Conclusion

This thesis deals with packet classification, which is integral part of many networking applications, most notably firewalls. Aim of the thesis is to design new algorithm that is applicable for 100 Gb/s networks and above. FPGAs and ASICs are considered as the target technology for the algorithm implementation.

All previous algorithms fail to achieve high throughput without the use of specialized and expensive TCAMs. Furthermore, most known algorithms suffer with the issue of non-deterministic throughput. Their worst case throughput may be significantly worse than in the average case. Detailed analysis of the previous algorithms is in Chapter 4. The family of decomposition methods appears to be the best candidate for the 100 Gb/s solution.

Analysis of several real life and synthetic rule sets gives the starting point for designing the new algorithm. The LPM stage, created by the problem decomposition, does not seem to be the biggest issue, because rule sets often contain only limited amount of distinct prefixes. More important is the efficient handling of LPM results - the Rule Mapping stage. After considering the ordinary hash function to map the LPM results into the rule table, a smarter approach is chosen. The perfect hash function construction algorithm is used to create direct mapping. The resulting function employs collision whenever it is suitable.

The new Perfect Hashing Crossproduct Algorithm [41] achieves the throughput of 266 million packets per second, which is well above the throughput of 100 Gb/s Ethernet for the shortest packets in one direction. This throughput is maintained under all circumstances, regardless the rule set size and the properties of the incoming packets. The high throughput is achieved also by effective mapping of the algorithm to hardware. Almost all parts of the algorithm are designed to fit into the FPGA or ASIC, where extremely high throughput can be achieved. The only access to the external memory is the perfect hash function evaluation and it takes exactly two narrow integer reads to classify each packet. This way the scarce external memory bandwidth is saved.

While the PHCA achieves very high throughput, its memory requirements are rather high. The Prefix Filtering Classification Algorithm [28] improves the memory efficiency with the throughput unchanged. The algorithm is based on the empirical observation that many rules define conditions only for several packet header fields, leaving other fields with the ANY value. These rules however significantly contribute to the final memory size of PHCA. By finding and applying generalization rules to the LPM results, the perfect hash table size is reduced by up to 94.9 %.

The disadvantage of PFCA is its low stability. The memory reduction is only 11 % for one of the used rule sets. The Prefix Coloring Classification Algorithm [40] is more general approach, which is not limited to one specific property of the rule set. After assigning colors

to prefixes, combinations of nonsense colors are found and avoided. This optimization lowers the size of the perfect hash table by an order of magnitude for most rule sets, while the throughput is still not affected – it is the same as in the original PHCA.

PCCA also brings several open questions. The most important question is whether there exists an optimal coloring strategy which always finds the best possible coloring in better than exponential time. The performed experiments show that multiple runs of the algorithm with random coloring give results similar to the normal distribution. Given the fact that all inputs to the algorithm are discrete and there is no concept of erroneous or random behavior (except for the coloring) which is often source of the normal distribution, it appears that some very complex behavior emerges in the algorithm. This behavior is yet to be fully understood.

Final algorithm is the MSPCCA, which combines PCCA with older MSCA. It takes the idea of dividing the rule set into several independent subsets from the MSCA. This technique brings significant reduction of memory. The average memory size of MSPCCA is by an order of magnitude smaller than in other algorithms. What is probably even more important is that MSPCCA shows high stability, keeping total memory requirements between 81 and 568 kbit for all available rule sets.

Throughput of all presented algorithms is 266 million packets per second, which corresponds to 178 Gb/s for the shortest 64 B packets and 548 Gb/s for the 440 B packets (reported as average in [36]).

11.1 Contributions

Contributions of the thesis are:

- Analysis of current classification algorithms.
- Statement of the algorithm attributes required for the 100 Gb/s throughput.
- Design of the Perfect Hashing Crossproduct algorithm. The algorithm maintains very high throughput under all conditions.
- Design of the Prefix Filtering Classification Algorithm, which is an improvement of the previous one in terms of the required memory size without compromising the speed.
- Design of the Prefix Coloring Classification Algorithm, which is yet another improvement of PHCA. PCCA is more general then PFCA and gives better results in most cases.
- Design of the Multi Subset Prefix Coloring Classification Algorithm, which is a combination of PCCA and older MSCA. MSPCCA maintains the throughput of PCCA and reduces the memory in all tested cases.
- Experimental evaluation of algorithm properties, mainly the memory requirements.

All four presented algorithms are part of the Netbench experimental framework [42]. The framework contains software models of the algorithms with the detailed reporting about the algorithm progress and results. It is designed for better understanding of the algorithms and easy modifications for further experiments.

The PHCA was implemented on the Xilinx Virtex-5 FPGA as the classification engine of the NIFIC firewall by the academic research association Cesnet in 2008-2010. [39, 14] The technology was transferred to the Invea-Tech company, where NIFIC is offered as a commercial product at the time of writing [3].

Bibliography

[1] Application Layer Packet Classifier for Linux. http://l7-filter.sourceforge.net, December 2011.

[2] Cisco asa 5500 series adaptive security appliances. http://www.cisco.com/en/US/products/ps6120/index.html, November 2011.

[3] Nific – wire-speed packet filtering and forwarding. http://www.invea-tech.com/products-and-services/nific-appliance, January 2012.

[4] Rldram part catalog. Micron Technology, Inc.

[5] Snort. http://www.snort.org, October 2011.

[6] Windows firewal. http://windows.microsoft.com/en-US/windows7/products/features/windows-firewall, November 2011.

[7] Florin Baboescu and George Varghese. Scalable packet classification. In *SIGCOMM '01: Proceedings of the 2001 conference on Applications, technologies, architectures, and protocols for computer communications*, pp. 199–210, New York, NY, USA, 2001. ACM. ISBN 1-58113-411-8.

[8] Burton Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 1970.

[9] Jean-Louis Brelet. An overview of multiple cam designs in virtex family devices. Xilinx Application Note 201, Xilinx Inc., 1999.

[10] Andrei Broder, Michael Mitzenmacher, Andrei Broder, and Michael Mitzenmacher. Network applications of bloom filters: A survey. In *Internet Mathematics*, pp. 636–646, 2002.

[11] B. Claise. Specification of the IP Flow Information Export (IPFIX) Protocol for the Exchange of IP Traffic Flow Information. RFC 5101, January 2008.

[12] Curtis R. Cook and R. R. Oldehoeft. A letter oriented minimal perfect hashing function. *SIGPLAN Not.*, 17:18–27, September 1982.

[13] Zbigniew J. Czech, George Havas, and Bohdan S. Majewski. An optimal algorithm for generating minimal perfect hash functions. *Information Processing Letters*, 43(5):257–264, 1992.

[14] Tomáš Dedek and Jan Kořenek. Hardware Packet Filter with IPv6 Support. CESNET technical report 26/2010, 2010.

[15] S. Dharmapurikar, P. Krishnamurthy, T. Sproull, and J. Lockwood. Deep packet inspection using parallel bloom filters. In *Proceedings of the 11th Symposium on High Performance Interconnects*, Aug 2003. ISBN 0-7695-2012-X.

[16] Sarang Dharmapurikar, Praveen Krishnamurthy, and David E. Taylor. Longest prefix matching using Bloom filters. In *SIGCOMM '03: Proceedings of the 2003 conference on Applications, technologies, architectures, and protocols for computer communications*, pp. 201–212, New York, NY, USA, 2003. ACM. ISBN 1-58113-735-4.

[17] Sarang Dharmapurikar, Haoyu Song, Jonathan Turner, and John Lockwood. Fast packet classification using Bloom filters. In *ANCS '06: Proceedings of the 2006 ACM/IEEE symposium on Architecture for networking and communications systems*, pp. 61–70, New York, NY, USA, 2006. ACM. ISBN 1-59593-580-0.

[18] M. Dietzfelbinger, A. Karlin, K. Mehlhorn, H. Rohnert, and R.E. Tarjan. Dynamic perfect hashing: upper and lower bounds. In *Foundations of Computer Science, 1988., 29th Annual Symposium on*, pp. 524 –531, oct 1988.

[19] M. Dietzfelbinger and F. Meyer. A new universal class of hash functions and dynamic hashing in real time. In Michael Paterson, editor, *Automata, Languages and Programming*, volume 443 of *Lecture Notes in Computer Science*, pp. 6–19. Springer Berlin / Heidelberg, 1990.

[20] Will Eatherton, George Varghese, and Zubin Dittia. Tree bitmap: hardware/software IP lookups with incremental updates. *SIGCOMM Computer Communication Review*, 34(2):97–122, 2004.

[21] P. García-Teodoro, J. Díaz-Verdejo, G. Maciá-Fernández, and E. Vázquez. Anomaly-based network intrusion detection: Techniques, systems and challenges. *Computers and Security*, 28(1-2):18 – 28, 2009.

[22] Pankaj Gupta. *Algorithms for Routing Lookups and Packet Classification*. PhD thesis, Stanford University, 2000.

[23] Pankaj Gupta and N. McKeown. Packet classification using hierarchical intelligent cuttings. In *Proc. Hot Interconnects*, 1999.

[24] Pankaj Gupta and Nick McKeown. Packet classification on multiple fields. In *SIGCOMM '99: Proceedings of the conference on Applications, technologies, architectures, and protocols for computer communication*, pp. 147–160, New York, NY, USA, 1999. ACM. ISBN 1-58113-135-6.

[25] G. Jaeschke. Reciprocal hashing: a method for generating minimal perfect hashing functions. *Commun. ACM*, 24:829–833, December 1981.

[26] Michal Kajan. Optimization of crossproduct-based classification algorithms. diploma thesis, FIT BUT, Brno, 2008.

[27] Jan Kořenek and Vlastimil Košař. Nfa split architecture for fast regular expression matching. In *Proceedings of the 6th ACM/IEEE Symposium on Architectures for Networking and Communications Systems*. Association for Computing Machinery, 2010. ISBN 978-1-4503-0379-8.

[28] Jan Kořenek, Viktor Puš, and Juraj Blaho. Memory optimization for packet classification algorithms. In *Proceedings of the 5th ACM/IEEE Symposium on Architectures for Networking and Communications Systems*, Association for Computing Machinery, pp. 165–166. Association for Computing Machinery, 2009. ISBN 978-1-60558-630-4.

[29] A. Kumar, Jun Xu, and Jia Wang. Space-code bloom filter for efficient per-flow traffic measurement. *IEEE Journal on Selected Areas in Communications*, 24(12), 2006.

[30] Sailesh Kumar, Balakrishnan Chandrasekaran, Jonathan Turner, and George Varghese. Curing regular expressions matching algorithms from insomnia, amnesia, and acalculia. In *Proceedings of the 3rd ACM/IEEE Symposium on Architecture for networking and communications systems*, ANCS '07, pp. 155–164, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-945-6.

[31] T. V. Lakshman and D. Stiliadis. High-speed policy-based packet forwarding using efficient multi-dimensional range matching. *SIGCOMM Comput. Commun. Rev.*, 28(4):203–214, 1998.

[32] H Lee, W Jiang, and V K Prasanna. Scalable High-Throughput SRAM-Based Architecture for IP Lookup Using FPGA. In *FPL '08*. IEEE, 2008.

[33] Ji Li, Haiyang Liu, and Karen Sollins. AFBV: a scalable packet classification algorithm. *SIGCOMM Comput. Commun. Rev.*, 32(3):24–24, 2002.

[34] Yi Lu, Balaji Prabhakar, and Flavio Bonomi. Perfect hashing for network applications. *Information Theory, 2006 IEEE International Symposium on*, pp. 2774–2778, July 2006.

[35] Steven McCanne and Van Jacobson. The bsd packet filter: a new architecture for user-level packet capture. In *Proceedings of the USENIX Winter 1993 Conference Proceedings on USENIX Winter 1993 Conference Proceedings*, pp. 2–2, Berkeley, CA, USA, 1993. USENIX Association.

[36] S. McCreary and K. Claffy. Trends in wide area IP traffic patterns - A view from Ames Internet Exchange. In *ITC Specialist Seminar*, Monterey, CA, Sep 2000.

[37] Mark H. Overmars and A. Frank van der Stappen. Range searching and point location among fat objects. *J. Algorithms*, 21(3):629–656, 1996.

[38] Rasmus Pagh and Flemming Friche Rodler. Cuckoo hashing. *Journal of Algorithms*, 51(2):122 – 144, 2004.

[39] Viktor Puš and Tomáš Dedek. Designing a Hardware-Accelerated Firewall with Two 10 Gbps Ports. CESNET technical report 8/2008, 2008.

[40] Viktor Puš, Michal Kajan, and Jan Kořenek. Hardware architecture for packet classification with prefix coloring. In *IEEE Design and Diagnostics of Electronic Circuits and Systems DDECS'2011*, pp. 231–236. IEEE Computer Society, 2011. ISBN 978-1-4244-9753-9.

[41] Viktor Puš and Jan Kořenek. Fast and scalable packet classification using perfect hash functions. In *FPGA '09: Proceedings of the 17th international ACM/SIGDA symposium on Field programmable gate arrays*, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-410-2.

[42] Viktor Puš, Jiří Tobola, Jan Kaštil, Vlastimil Košař, and Jan Kořenek. Netbench - the framework for evaluation of packet processing algorithms. In *Proceedings of the 7th ACM/IEEE Symposium on Architectures for Networking and Communications Systems*, pp. 95–96. IEEE Computer Society, 2011. ISBN 978-0-7695-4521-9.

[43] Sumeet Singh, Florin Baboescu, George Varghese, and Jia Wang. Packet classification using multidimensional cutting. In *SIGCOMM '03: Proceedings of the 2003 conference on Applications, technologies, architectures, and protocols for computer communications*, pp. 213–224, New York, NY, USA, 2003. ACM. ISBN 1-58113-735-4.

[44] Haoyu Song and John W. Lockwood. Efficient packet classification for network intrusion detection using FPGA. In *FPGA '05: Proceedings of the 2005 ACM/SIGDA 13th international symposium on Field-programmable gate arrays*, pp. 238–245, New York, NY, USA, 2005. ACM. ISBN 1-59593-029-9.

[45] Haoyu Song, Jonathan Turner, and John Lockwood. Shape shifting tries for faster ip route lookup. In *ICNP '05: Proceedings of the 13TH IEEE International Conference on Network Protocols*, pp. 358–367, Washington, DC, USA, 2005. IEEE Computer Society. ISBN 0-7695-2437-0.

[46] V. Srinivasan, G. Varghese, S. Suri, and M. Waldvogel. Fast and scalable layer four switching. *SIGCOMM Comput. Commun. Rev.*, 28(4):191–202, 1998.

[47] D. Taylor and J. Turner. Scalable packet classification using distributed crossproducing of field labels. In *IEEE INFOCOM 2005, 24th Annual Joint Conference of the IEEE Computer and Communications Societies.*, pp. 269–280, July 2005.

[48] David E. Taylor and Jonathan S. Turner. Classbench: a packet classification benchmark. *IEEE/ACM Trans. Netw.*, 15(3):499–511, 2007.

[49] B. Trammell, E. Boschi, L. Mark, T. Zseby, and A. Wagner. Specification of the IP Flow Information Export (IPFIX) File Format. RFC 5655, October 2009.

[50] Balajee Vamanan, Gwendolyn Voskuilen, and T. N. Vijaykumar. Efficuts: optimizing packet classification for memory and throughput. In *Proceedings of the ACM SIGCOMM 2010 conference on SIGCOMM*, SIGCOMM '10, pp. 207–218, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0201-2.

[51] Pavel Čeleda and Radek Krejčí. An analysis of the chuck norris botnet 2. http://www.muni.cz/ics/research/cyber/chuck_norris_botnet, October 2011.

[52] Pavel Čeleda, Radek Krejčí, Jan Vykopal, and Martin Drašar. Embedded malware - an analysis of the chuck norris botnet. In *Proceedings of the 2010 European Conference on Computer Network Defense*, EC2ND '10, pp. 3–10, Washington, DC, USA, 2010. IEEE Computer Society. ISBN 978-0-7695-4311-6.